UNIVERSITÉ DE MONTRÉAL

TRAÇAGE ET PROFILAGE D'APPLICATIONS D'APPRENTISSAGE AUTOMATIQUE DE TYPE FLOT DE DONNÉES UTILISANT UN PROCESSEUR GRAPHIQUE

PIERRE ZINS
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2018

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

TRAÇAGE ET PROFILAGE D'APPLICATIONS D'APPRENTISSAGE AUTOMATIQUE DE TYPE FLOT DE DONNÉES UTILISANT UN PROCESSEUR GRAPHIQUE

présenté par : ZINS Pierre
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. ANTONIOL Giuliano, Ph. D., président
M. DAGENAIS Michel, Ph. D., membre et directeur de recherche
M. ALOISE Daniel, Ph. D., membre

DÉDICACE

À mes parents, mon frère et ma famille pour leur grande confiance
et leur soutien sans faille tout au long de mes études,
malgré la distance.

REMERCIEMENTS

Je tiens tout d'abord à remercier mon directeur de recherche Michel Dagenais qui m'a accompagné tout au long de ce projet. Son suivi constant, sa disponibilité, ses connaissances techniques ainsi que ses nombreux conseils ont été très précieux pour l'avancement du projet.

Je souhaite également remercier les deux associés de recherche, Geneviève et Naser, pour leurs conseils et les nombreuses aides qu'ils ont pu apporter. Leurs très bonnes connaissances et leur expérience dans le domaine ont constitué une aide significative pour mon travail.

Merci ensuite à tous mes collègues et amis du laboratoire DORSAL pour leur bonne humeur et les agréables moments partagés. Cela a représenté un élément clé pour l'avancement de mon travail. De même, je remercie toutes les personnes avec qui j'ai pu discuter à propos de mon projet de recherche et qui m'ont permis d'avoir des conseils et commentaires quant à mon travail.

Enfin, je souhaite également souligner le support financier de la part des partenaires industriels : Ericsson, EfficiOS, Ciena, Google, Prompt et le Conseil de Recherches en Sciences Naturelles et en génie du Canada (CRSNG). Merci également à AMD pour leur don de matériel ainsi que leur disponibilité qui ont rendu possible ce projet de recherche.

RÉSUMÉ

Actuellement, les besoins en puissance de calcul sont de plus en plus importants, alors que les améliorations au niveau du matériel commencent à ralentir. La puissance des processeurs et notamment leur fréquence de fonctionnement stagnent pour des raisons physiques comme la finesse de gravure ou la dissipation de chaleur.

Afin de surpasser ces limites, le calcul en parallèle semble être une solution prometteuse avec l'utilisation d'architectures hétérogènes. Ces dernières mettent en œuvre une combinaison de plusieurs unités de calculs de types possiblement différents, ce qui leur permet d'offrir un fonctionnement hautement parallèle. Malgré tout, utiliser l'ensemble du matériel efficacement reste difficile, et la programmation au niveau logiciel de ces architectures pose problème. Par conséquent, différents modèles ont émergé avec notamment les approches flot de données. Ces dernières proposent des caractéristiques très adaptées pour ce genre de contexte parallèle. Elles permettent de programmer plus facilement les différentes unités de calcul afin de bénéficier au maximum du matériel disponible.

Dans un contexte de recherche de performance optimale, il est essentiel d'avoir des outils permettant de diagnostiquer d'éventuels problèmes. Quelques solutions ont déjà pu démontrer leur efficacité dans le cas d'un modèle de programmation plus traditionnel et séquentiel, utilisant ou non un processeur graphique. On retrouve par exemple des outils comme *LTTng* ou *Ftrace* destinés à l'analyse du processeur central. Concernant les processeurs graphiques, les outils propriétaires et à sources fermées, proposés par les constructeurs sont en général les plus complets et privilégiés par les programmeurs. Cela présente toutefois une limite, puisque les solutions ne sont pas générales et restent dépendantes du matériel proposé par un constructeur. Par ailleurs, elles offrent une flexibilité limitée avec des visualisations et analyses définies et fixes qui ne peuvent ni être modifiées ni améliorées en fonction des besoins d'un utilisateur. Finalement, aucun outil existant ne s'intéresse spécifiquement aux modèles flot de données.

Dans le cadre de ce projet de recherche, nous essayons donc de répondre à ces manques. Nous cherchons à proposer un outil ou une technique pour le diagnostic et l'analyse de performance destiné aux applications basées sur des approches flot de données et s'exécutant sur une architecture hétérogène. Nous nous concentrons sur le cas où le processeur traditionnel est accompagné par un processeur graphique afin d'accélérer certaines tâches impliquant beaucoup de calculs sur un grand ensemble de données. Notre travail est basé sur des techniques de traçage et de profilage et a pour objectif de rester relativement général. Ainsi,

même si l'implémentation proposée est destinée à la bibliothèque logicielle TensorFlow, il serait envisageable de l'appliquer à d'autres cas. TensorFlow est l'un des outils phare pour l'apprentissage automatique et propose un fonctionnement basé sur un modèle flot de données avec un graphe de calcul. Cette bibliothèque est évidemment destinée à une utilisation sur une architecture hétérogène avec notamment un ou plusieurs processeurs graphiques. Par ailleurs, un mode de fonctionnement distribué est aussi proposé et sera étudié. TensorFlow supporte de manière officielle les processeurs graphiques Nvidia mais plusieurs travaux visent à étendre cela à d'autres constructeurs. On note une première solution basée sur la plateforme ROCm de AMD. La seconde option concerne la spécification SYCL qui propose une couche d'abstraction haut niveau au-dessus d'OpenCL afin de programmer un processeur graphique.

La technique proposée est générale et fonctionne avec les trois possibilités offertes par TensorFlow pour utiliser un processeur graphique. Elle permet de collecter un grand nombre de données à propos de divers éléments et de les combiner dans une trace. Par la suite, un traitement de cette dernière ainsi que plusieurs méthodes de visualisations permettent de dégager des informations intéressantes à propos de l'exécution d'une application. À partir de ces résultats, une analyse de la performance est possible pour détecter une utilisation inefficace du matériel disponible, des éléments limitants ou des parties propices à des optimisations. Dans ce mémoire, nous présentons différents cas pour lesquels les résultats fournis à l'utilisateur ont permis d'optimiser une application et de réduire le temps d'exécution. Finalement, nous nous assurons que le surcoût introduit par notre technique reste raisonnable et ne compromet pas son utilisation.

ABSTRACT

Recently, increasing computing capabilities have been required in various areas like scientific computing, video games and graphical rendering or artificial intelligence. These domains usually involve the processing of a large amount of data, intended to be performed as fast as possible. Unfortunately, hardware improvements have recently slowed down. The CPU clock speed, for example, is not increasing much any more, possibly nearing technological limits. Physical constraints like the heat dissipation or fine etching are the main reasons for that.

Consequently, new opportunities like parallel processing using heterogeneous architectures became popular. In this context, the traditional processors get support from other computing units like graphical processors. In order to program these, the dataflow model offers several advantages. It is inherently parallel and thus well adapted.

In this context, guaranteeing optimal performances is another main concern. For that, tracing and profiling central processing and graphical processing units are two useful techniques that can be considered. Several tools exist, like *LTTng* and *FTrace* that can trace the operating system and focus on the central processor. In addition, proprietary tools offered by hardware vendors can help to analyze and monitor the graphical processor. However, these tools are specific to one type of hardware and lack flexibility. Moreover, none of them target in particular dataflow applications executed on a heterogeneous platform.

Through this research project, our objective is to provide a very flexible and complete performance analysis environment with information related to the traditional processor, the graphical processor, the operating system and also the dataflow model. We evaluate our solution with TensorFlow, a popular machine learning and deep learning library that uses a dataflow model for the execution. In our project, we focused on this library but our work remains generally applicable, and the implementation for another library is possible.

The proposed method is based on collecting information during the execution of an application by using tracing and profiling. After this step, we obtain a trace that contains numerous events. Then, the trace is post-processed and analyzed. Graphical visualizations are offered to the user and help to understand the execution of an application as well as its performance.

To demonstrate the efficiency of our proposed technique, we present several examples in which the information collected, the analyses and the visualizations offered are helpful for the user. We show that we are able to optimize a TensorFlow application by using the results of our tracing and profiling method. Most of the time, it consists in an execution time or

memory usage reduction. Finally, we also measure the introduced overhead to demonstrate that it stays reasonable and does not represent a serious drawback for our work.

TABLE DES MATIÈRES

DÉDICACE iii

REMERCIEMENTS iv

RÉSUMÉ v

ABSTRACT vii

TABLE DES MATIÈRES ix

LISTE DES TABLEAUX xii

LISTE DES FIGURES xiii

LISTE DES SIGLES ET ABRÉVIATIONS xvii

CHAPITRE 1 INTRODUCTION 1
1.1 Définitions et concepts de base 1
1.1.1 Processeur graphique et processeur central 1
1.1.2 Traçage, instrumentation et profilage 2
1.1.3 Visualisation des résultats 2
1.2 Éléments de la problématique 3
1.3 Objectifs de recherche 5
1.3.1 Questions de recherche 5
1.3.2 Objectifs spécifiques 5
1.3.3 Hypothèse de recherche 5
1.4 Plan du mémoire 6

CHAPITRE 2 REVUE DE LITTÉRATURE 7
2.1 Modèle flot de données 7
2.1.1 Principes théoriques 7
2.1.2 Langages de type flot de données 8
2.1.3 Méthodes et techniques flot de données 9
2.1.4 Modèle flot de données appliqué à l'apprentissage automatique 10
2.2 Architectures hétérogènes et processeurs graphiques 14
2.2.1 Architectures hétérogènes 14

2.2.2 Historique . . . 15
2.2.3 Architectures . . . 16
2.2.4 Librairies pour le calcul parallèle sur GPU . . . 18
2.2.5 OpenCL . . . 21
2.2.6 HSA . . . 22
2.2.7 HIP . . . 22
2.3 Traçage et profilage . . . 23
2.3.1 Traçage et profilage du processeur traditionnel . . . 23
2.3.2 Traçage et profilage du processeur graphique . . . 27
2.4 Analyse et visualisation des résultats . . . 37
2.4.1 Babeltrace . . . 37
2.4.2 Trace Compass . . . 38
2.5 Profilage d'applications flot de données . . . 39
2.6 Conclusion de la revue de littérature . . . 41

CHAPITRE 3 MÉTHODOLOGIE . . . 43
3.1 Utilisation de TensorFlow avec un processeur graphique . . . 43
3.2 Environnement de travail . . . 45
3.3 Traçage et profilage avec la technique proposée . . . 46
3.4 Code source du travail . . . 48

CHAPITRE 4 ARTICLE 1 : TRACING AND PROFILING MACHINE LEARNING DATAFLOW APPLICATIONS ON GPU . . . 49
4.1 Introduction . . . 49
4.2 Related work . . . 52
4.2.1 CPU tracing and profiling . . . 52
4.2.2 GPU tracing and profiling . . . 53
4.2.3 Dataflow profiling . . . 54
4.3 Background . . . 55
4.3.1 TensorFlow concepts . . . 55
4.3.2 TensorFlow with a GPU . . . 56
4.4 Proposed method . . . 56
4.4.1 Architecture . . . 57
4.4.2 Multilevel Data Collection . . . 57
4.4.3 Trace Correlation and Analysis . . . 66
4.4.4 Visualization . . . 68
4.5 Use cases . . . 72

4.5.1 Triplet loss example . . . 72
4.5.2 Compute-bound example . . . 76
4.5.3 Distributed Dataflow graph . . . 79
4.5.4 Inference example . . . 83
4.5.5 Memory management example . . . 85
4.6 Evaluation . . . 87
4.6.1 Setup . . . 89
4.6.2 Overhead analysis and discussion . . . 89
4.7 Conclusion and future work . . . 94

CHAPITRE 5 DISCUSSION GÉNÉRALE . . . 95
5.1 Cas d'utilisation . . . 95
5.2 Utilisation . . . 95
5.3 Performance . . . 96
5.4 Contributions additionnelles . . . 96

CHAPITRE 6 CONCLUSION . . . 98
6.1 Synthèse des travaux . . . 98
6.2 Limitations de la solution proposée . . . 99
6.3 Améliorations futures . . . 100

RÉFÉRENCES . . . 102

LISTE DES TABLEAUX

Tableau 3.1	Configuration matérielle	45
Tableau 3.2	Configuration logicielle	46
Table 4.1	Most demanding nodes of the graph. Several of them are related to the MaxPool operations.	77
Table 4.2	Hardware and software configuration	89
Table 4.3	Benchmark : Autoencoder - ROCm platform	90
Table 4.4	Benchmark : Autoencoder - CUDA	91
Table 4.5	Benchmark : Autoencoder - SYCL	92
Table 4.6	Benchmark : Convolutional Neural Network - ROCm platform	92
Table 4.7	Benchmark : Convolutional Neural Network - CUDA	93
Table 4.8	Benchmark : Convolutional Neural Network - SYCL	93

LISTE DES FIGURES

Figure 2.1 Exemple de modèle flot de données. Les données de départ sont fournies en entrée aux nœuds 1 et 3. Elles circulent ensuite le long des arcs et sont traitées par chaque nœud. Le résultat final correspond à la sortie du nœud 8. 8

Figure 2.2 Tensorboard : visualisation du graphe d'un réseau de neurones ayant deux couches cachées et une couche de sortie. Une partie annexe du graphe va gérer la propagation arrière et le calcul des gradients mais n'est pas représentée ici. 11

Figure 2.3 Exemple de programme TensorFlow 13

Figure 2.4 Graphe de calcul correspondant au code précédent 13

Figure 2.5 Modèle d'exécution de CUDA. La grille et les différents blocs peuvent avoir 1, 2 ou 3 dimensions. 19

Figure 2.6 Modèle de mémoire utilisé par CUDA 20

Figure 2.7 Exemple de résultat de profilage avec `nvprof`. On retrouve une ligne pour chaque opération effectuée sur le processeur graphique (noyau de calcul, copie de données). Plusieurs statistiques sont disponibles : le nombre total d'appels, la durée totale, la durée moyenne, les durées maximale et minimale. 28

Figure 2.8 Graphique temporel de l'exécution d'une application. Cette visualisation est disponible avec `Nvidia Visual Profiler`. On retrouve plusieurs lignes représentant les appels de fonction à l'API CUDA, les transferts de mémoire, les noyaux de calculs exécutés par le GPU ainsi que les streams (ou files) dans lesquelles les opérations destinées au processeur graphique ont été soumises. 29

Figure 2.9 L'analyse automatique de `Nvidia Visual Profiler` propose un classement des noyaux de calcul en fonction de l'importance de leur optimisation. Les critères utilisés sont la durée d'exécution ainsi que le taux d'occupation atteint par chaque noyau. Ainsi, les noyaux les mieux classés sont les plus intéressants à optimiser afin de gagner en performance. 29

Figure 2.10 Graphique présentant le nombre d'exécution de chaque type d'instruction sous la forme de pourcentage. Cela concerne l'exécution d'un unique noyau de calcul. 30

Figure 2.11 Graphique proposé par `Nvidia Visual Profiler` montrant l'utilisation de l'unité de calcul et de mémoire. Il permet d'expliquer à l'utilisateur si l'exécution d'un noyau de calcul est limitée par les calculs ou par la mémoire. 30
Figure 2.12 Information détaillée à propos de l'utilisation des différents composants de mémoire (mémoire partagée, cache L2, cache unifié, mémoire globale). 31
Figure 2.13 Vue temporelle CodeXL. On peut suivre l'ensemble des appels de fonction de l'API HSA, les noyaux de calculs exécutés ainsi que les transferts de données entre la mémoire centrale et celle de la carte graphique. 33
Figure 2.14 Vue temporelle du profilage d'une application ayant été instrumentée avec les marqueurs AMD. L'ensemble des événements liés à l'instrumentation statique sont présentés en orange. 34
Figure 2.15 Vue présentant les compteurs de performance sous la forme d'un tableau. Chaque ligne montre différentes métriques à propos de l'exécution d'un noyau de calcul. 35
Figure 2.16 Exemple de trace du noyau Linux générée avec LTTng et visualisée avec Babeltrace. 38
Figure 4.1 Performance analysis architecture. Information about different components is collected at several levels. The resulting trace is post-processed and then the results can be statistically analyzed or graphically visualized. 57
Figure 4.2 Distributed dataflow example : the graph is shared among two machines and data is fed to two input nodes (A and F). The computation of node E (assigned to machine 1) needs the result of node G (assigned to machine 2). Therefore, a tensor exchange between the machines is required. 62
Figure 4.3 OpenCL profiling : Several events happen on both devices and the corresponding timestamps can be collected and used for synchronization 68
Figure 4.4 Example of matching of the GPU kernels with the nodes from the computation graph 68
Figure 4.5 Initial application : the beginning of the 88^{th} execution of the graph is shown. The GPU waits for a significant time before starting its job, and the CPU seems to inefficiently process some nodes during this time. 73
Figure 4.6 CPU-assigned pre-processing operations : The CPU processes many times these two sequences of nodes before the GPU starts. 73

Figure 4.7 Execution with the first optimization : setting *num_parallel_calls* to 8. The processing done on the CPU has been parallelized and is more efficient now. Significant time was saved. 74
Figure 4.8 Execution with the two optimizations : setting *num_parallel_calls* to 8 and *prefetch* to 1. In addition to the parallelization of CPU work, pipelining has been added and increases again the performance. . . . 75
Figure 4.9 Initial execution with the convolution and maxpool operations. The GPU seems to be in use all the time. 76
Figure 4.10 Evolution of the accuracy and the loss after 1000 iterations. The accuracy loss caused by the use of strided convolutions with no maxpool layer is subtle. 78
Figure 4.11 Execution with the strided convolution. The GPU is still used all the time, but the duration of 3 executions of the graph has been reduced. 79
Figure 4.12 Distributed execution of a CNN on 2 machines - first experiment. Machine 1 executes the first convolutional layer and the machine 2 asks for the result. Once received, machine 2 can compute the second convolutional layer and the dense layer. The XY chart at the bottom shows the size of the tensors exchanged between the machines. 80
Figure 4.13 Division of the graph for both experiments. 81
Figure 4.14 Distributed execution of a CNN on 2 machines - second experiment. This time, machine 1 executes the two convolutional layers and machine 2 asks for the result. Once received, machine 2 can compute the dense layer. The XY chart at the bottom shows that the size of the tensors exchanged between the machines is reduced compared to the first experiment. 82
Figure 4.15 Initial Callstack and Linux Kernel memory usage views for an inference step. A long time is spent before the beginning of the graph execution and around 800 MB of RAM memory is used by the TensorFlow application. 83
Figure 4.16 Kernel traces of the inference step. We can detect a lot of *pread* system calls. They correspond to the loading of the model file. 84
Figure 4.17 Callstack and kernel status, with the static instrumentation of the *LoadGraph* function. We can verify that this function is indeed time consuming. 84

Figure 4.18 Kernel traces, Callstack and Linux Kernel memory usage view of the optimized application. Using the memory mapping technique to load the model file decreases the duration of the *LoadGraph* function. The RAM memory usage is also decreased at the beginning of the graph execution, compared to the first version of the application. 85

Figure 4.19 Normal execution using feed_dict argument. An important percentage of the execution is dedicated to a large memory transfer from the RAM memory to the GPU memory. 86

Figure 4.20 After the optimization with a slice operation and a unique large memory copy. Now, the whole dataset is copied to the GPU memory at the application initialization. Consequently, the data batches are already in the GPU memory for all the graph executions. Therefore, each execution is much faster and consists mostly in GPU kernels processing. 87

LISTE DES SIGLES ET ABRÉVIATIONS

ADN	DNA - Deoxyribonucleic acid
AMD	Advanced Micro Devices
API	Application Programming Interface
APU	Accelerated Processing Unit
ARM	Advanced RISC Machine
AVC	Advanced Video Coding
BLAS	Basic Linear Algebra Subprograms
CAL	Cal Actor Language
clBLAS	OpenCL Basic Linear Algebra Subprograms
CLUST	OpenCL User Space Tracepoint
CPU	Central Processing Unit
CTF	Common Trace Format
CUDA	Compute Unified Device Architecture
CUPTI	CUDA Profiling Tools Interface
DMA	Direct Memory Access
DSP	Digital Signal Processor
Factorisation LU	Factorisation Lower Upper
FPGA	Field Programmable Gate Array
GCN	Graphical Cores Next
GPA	GPU Performance API
GNU	GNU's Not Linux
GPU	Graphical Processing Unit
HC	Heterogeneous Compute
HCC	Heterogeneous Compute Compiler
HDL	Hardware Description Language
HEVC	High Efficiency Video Coding
HIP	Heterogeneous-Compute Interface for Portability
HSA	Heterogeneous System Architecture
IGP	Integrated Graphic Processor
ISA	Instruction Set Architecture
JPEG	Joint Photographic Experts Group
kprobe	Kernel probe
GPGPU	General Purpose Graphical Processing Unit

LTTng	Linux Tracing Toolkit next generation
MATLAB	Matrix Laboratory
MPEG	Moving Picture Experts Group
MPI	Message Pssing Interface
NVPROF	Nvidia Visual Profiler
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
OpenMP	Open Multi-Processing
OS	Operating System
OTF	Open Trace Format
RAM	Random-Access Memory
RCP	Radeon Compute Profiler
RCV	Reconfigurable Video Coding
RCV-CAL	Reconfigurable Video Coding - Cal Actor Language
RDF	Resource Description Framework
ROCm	Radeon Open Compute platform
SCADE	Safety-Critical Application Development Environment
SIMD	Single Instruction Multiple Data
TAU	Tuning and Analysis Utilities
TPU	Tensor Processing Unit
TraceFS	Trace File System
uprobe	user probe
USDT	User Statically Defined Tracing
UST	User-Space Tracing
VHDL	VHSIC Hardware Description Language
XML	Extensible Markup Language

CHAPITRE 1 INTRODUCTION

Au cours des dernières années, les besoins en termes de puissance de calcul n'ont cessé d'augmenter. De nombreux domaines comme le calcul scientifique, le rendu graphique en plusieurs dimensions ou encore l'apprentissage automatique requièrent des performances toujours plus grandes afin de traiter un maximum de données en un minimum de temps. Par ailleurs, les améliorations matérielles ont peu à peu stagné. Il n'est plus possible, par exemple, de continuellement augmenter le nombre de transistors sur un microprocesseur afin d'accroître sa fréquence, ou encore d'améliorer indéfiniment la finesse de gravure.

Pour remédier à cela, le calcul en parallèle a été considéré et permet d'augmenter les performances. Par conséquent, le principe d'architectures hétérogènes a émergé et combine plusieurs unités de calcul afin de mettre en place ce parallélisme. Un cas très fréquent concerne l'utilisation d'un processeur graphique permettant d'assister le processeur traditionnel pour les tâches très exigeantes en termes de calcul. La programmation de ce genre d'architecture est souvent complexe et difficile à mettre en place avec une approche traditionnelle et séquentielle. Par conséquent, d'autres possibilités ont été envisagées comme le modèle flot de données qui offre des caractéristiques avantageuses.

Par ailleurs, assurer une performance optimale est un point essentiel lors du développement d'une application. Pour cela, on retrouve notamment des techniques de traçage et de profilage. Ces dernières consistent à collecter de nombreux événements durant l'exécution d'une application afin de former une trace. Le traitement, l'analyse ainsi que la visualisation de cette dernière permettent à l'utilisateur de comprendre la performance d'une application et d'identifier d'éventuels problèmes ou limitations. Des outils permettant cela existent et ont déjà prouvé leur efficacité dans un cadre d'exécution plus traditionnel. Cependant, dans ce mémoire, nous nous intéressons aux architectures hétérogènes utilisant une approche de calcul par flot de données. Nous verrons dans les sous-sections 2.3 et 2.5 que les outils existants montrent des limites dans ce genre de contexte et nous chercherons à les combler au cours du projet de recherche.

1.1 Définitions et concepts de base

1.1.1 Processeur graphique et processeur central

Le processeur central correspond au composant traditionnel utilisé dans un ordinateur pour le calcul. Dans ce mémoire, différents termes seront utilisés pour le désigner : **processeur**

central, **processeur traditionnel** ou encore l'acronyme **CPU** correspondant au terme anglophone **Central Processing Unit**.

Le processeur graphique quant à lui correspond à une autre unité de calcul, destinée à l'origine aux parties graphiques des applications et à l'affichage sur l'écran. Récemment, le concept de **GPGPU** (**General Purpose GPU**) a émergé et consiste à utiliser ce composant pour toute tâche impliquant beaucoup de calcul sur un grand nombre de données. En effet, un processeur graphique est basé sur le concept de **SIMD (Single instruction, multiple data)** de la taxonomie de Flynn et consiste à appliquer une même instruction à plusieurs données simultanément. Cela est rendu possible par son architecture composée d'un très grand nombre de cœurs de calcul en comparaison d'un processeur traditionnel. Dans la suite du mémoire, on retrouve ce composant sous les termes suivants : **processeur graphique**, **carte graphique** ou encore **GPU (Graphical Processing Unit)**.

1.1.2 Traçage, instrumentation et profilage

Le traçage est une technique très utilisée dans le cadre de l'analyse de performance. Elle désigne le fait de collecter un grand nombre d'événements durant l'exécution d'une application sans dégrader de manière significative sa performance. Ces événements formeront une trace qui pourra être analysée et visualisée afin d'en faire ressortir des informations bénéfiques. Le traçage fonctionne souvent de pair avec une instrumentation qui consiste à insérer des points de trace à des endroits spécifiques dans le code source d'une application. Chaque point de trace atteint lors de l'exécution générera un événement en sortie. Une instrumentation demande généralement l'accès au code source de l'application, ou de la bibliothèque logicielle, et nécessite la recompilation de cette dernière. Ainsi, il sera possible de tracer au niveau utilisateur. Le traçage du noyau du système d'exploitation est également disponible et se base sur une instrumentation existante dans le noyau Linux.

Le profilage quant à lui correspond davantage à la collecte de mesures et métriques à propos de l'exécution d'une application. On peut notamment évoquer la récupération de compteurs de performance à propos du matériel ou encore le suivi de la consommation mémoire d'une application.

1.1.3 Visualisation des résultats

Le résultat d'une opération de traçage ou de profilage consiste généralement en une trace contenant des événements. En raison de leur très grand nombre, il n'est pas envisageable d'analyser simplement une liste d'événements présentée sous un format texte. Il est donc

essentiel de proposer des méthodes de visualisation graphiques. Trace Compass est une des solutions privilégiées pour cela et fonctionne avec des vues qui vont présenter de manière graphique les résultats à l'utilisateur. Chacune d'entre elles se concentrera sur des aspects précis et permettra de faire ressortir des informations utiles pour l'utilisateur. Dans ce mémoire, quatre types de vues sont abordés.

— La vue **Callstack** permet notamment de représenter l'imbrication des différents appels de fonctions réalisés par chaque fil d'exécution.
— La vue **Contrôle de flux** ou **Control Flow View** représente l'ensemble des fils d'exécution au niveau du noyau du système d'exploitation. Elle permet de suivre leur activité avec notamment les appels système, les blocages, les interruptions, le temps passé en mode utilisateur, les temps d'attentes, ...
— La vue des **Ressources** ou **Resources View** représente quant à elle l'état des différents cœurs du processeur central au cours de la session de traçage.
— Les **graphiques XY** ou **XY charts** sont plus adaptés à d'autres cas qui consistent à suivre l'évolution d'une certaine métrique au cours de l'exécution d'une l'application. Il peut par exemple s'agir de la consommation mémoire de l'application.

1.2 Éléments de la problématique

Actuellement, des outils permettant l'analyse de performance existent pour les processeurs centraux et graphiques. Cependant, de nombreux problèmes persistent dans le cas d'analyses réalisées pour des applications s'exécutant sur des plateformes hétérogènes.

Tout d'abord, les outils de traçage et de profilage existants se concentrent généralement sur un type de processeur. On peut penser à `Ftrace`, `Perf` ou `LTTng` qui sont à l'origine destinés au processeur central. En ce qui concerne les processeurs graphiques, les outils les plus utilisés restent ceux proposés par les constructeurs comme `Nsight` pour Nvidia ou `CodeXL` pour AMD. Ces derniers proposent des éléments très intéressants pour l'analyse de performance, mais uniquement du point de vue du processeur graphique. Ainsi, les outils existants peinent à faire le lien entre l'activité de ce dernier et celle du processeur traditionnel. Or, dans le cadre d'architectures hétérogènes, il est justement essentiel d'analyser l'ensemble des unités de calcul disponibles ainsi que leur coopération. Les transferts de mémoire entre les processeurs peuvent notamment affecter grandement la performance. Par ailleurs, ces outils de profilage pour les cartes graphiques manquent généralement de flexibilité, ne supportent pas du matériel de différents constructeurs et ne s'intègrent pas bien dans un environnement d'analyse de performance complet.

Au travers de ce projet de recherche, nous viserons donc à solutionner ces problèmes en proposant des outils ou méthodes mieux adaptés. De nombreux problèmes concernant l'utilisation d'un processeur graphique ont été rencontrés, comme la gestion des opérations effectuées de manière asynchrone par celui-ci et la différence d'horloges entre les unités de calcul. En effet, le principe de fonctionnement d'un processeur graphique s'articule autour d'une ou plusieurs queues dans lesquelles les tâches lui sont soumises. Ces dernières sont ordonnancées par la carte graphique elle-même et généralement exécutées de manière asynchrone. Par conséquent, les événements correspondants sont collectés a posteriori et il faut veiller à les réordonner au sein de la trace finale. Par ailleurs, les temps récupérés pour le processeur graphique n'ont généralement aucun sens du point de vue du processeur central, puisque les horloges utilisées par les deux composants sont différentes.

Par ailleurs, il sera aussi nécessaire de relier les informations de bas niveau à propos de l'exécution avec d'autres à un niveau d'abstraction plus élevé, concernant le modèle flot de données lui-même. Actuellement, aucun outil pour l'analyse de performance ne permet cela. D'autres défis sont inhérents à l'approche flot de données. Il faudra s'assurer de pouvoir combiner ensemble toutes les informations collectées afin d'analyser la totalité du matériel utilisé. De plus, le travail effectué par le processeur graphique dans ce contexte est souvent important, de nombreux noyaux de calculs sont exécutés et parfois un grand nombre de fois. Il sera par conséquent essentiel de gérer toutes ces données de manière adéquate.

Un dernier défi provenait de notre volonté d'implémenter notre solution pour chacune des versions de TensorFlow qui permettait d'utiliser un processeur graphique. Nous avons donc considéré les trois solutions principales pour l'utilisation d'une carte graphique avec TensorFlow (CUDA, ROCm et SYCL). Chacune d'entre elles comprenait certains éléments spécifiques, ce qui a complexifié le développement de notre implémentation.

Au cours du projet de recherche, nous avons eu besoin d'un outil pour le traçage et l'instrumentation de code source. Le choix de `LTTng` s'est fait naturellement pour différentes raisons. En premier lieu, sa performance était très attirante. De plus, cet outil est relativement simple à utiliser et propose un résultat sous la forme d'une trace au format CTF. Ce format offre un avantage du point de vue de la performance, par un stockage compact de données, et bénéficie d'un environnement complet et flexible pour l'analyse et la visualisation de trace. Un autre avantage est la possibilité de collecter des traces concernant le noyau Linux.

1.3 Objectifs de recherche

Il existe déjà des outils permettant le traçage et profilage de processeurs traditionnels et graphiques. Cependant, ces derniers permettent rarement de considérer de manière conjointe les deux unités de calculs. Par ailleurs, les analyses de systèmes ou d'applications flot de données se limitent généralement à une analyse d'assez haut niveau, sans fournir d'information précise à propos du matériel, du système d'exploitation ou des bibliothèques logicielles intermédiaires. L'objectif principal de ce projet de recherche est donc de proposer une méthode de traçage et de profilage complète, à différents niveaux, et visant les applications flot de données qui s'exécutent sur des architectures hétérogènes.

1.3.1 Questions de recherche

Afin de remplir, l'objectif principal nous répondons à trois questions de recherche :

1. Quelles informations permettraient de comprendre la performance d'une application flot de données s'exécutant sur une architecture hétérogène ?
2. Quels méthodes et outils pourraient être utilisés conjointement afin de collecter ces informations ?
3. Quels types de visualisation et analyses permettraient de présenter les résultats de manière adéquate à l'utilisateur dans un objectif final d'optimisation de la performance ?

1.3.2 Objectifs spécifiques

Afin de répondre à l'objectif principal de ce projet de recherche, des objectifs spécifiques ont été énoncés.

1. Démontrer l'efficacité des méthodes de traçage et profilage pour la détection de problèmes de performance ou la suggestion d'optimisations dans le cadre d'applications de type flot de données bénéficiant de l'accélération des processeurs graphiques.
2. Développer des outils ou méthodes de traçage adaptés à la bibliothèque logicielle TensorFlow.
3. Développer des vues adéquates, présentant de manière efficace, détaillée et claire l'exécution d'une application flot de données sur l'ensemble du matériel disponible.

1.3.3 Hypothèse de recherche

Notre projet de recherche repose sur l'hypothèse suivante : Les méthodes de traçage et profilage permettent de détecter des problèmes de performance ainsi que d'identifier les éléments

les plus propices à des optimisations pour des applications flot de données s'exécutant sur des systèmes hétérogènes. L'originalité réside dans le fait qu'il n'existe pas d'outil permettant d'avoir une vue claire, complète et détaillée de l'exécution du programme flot de données sur un système hétérogène combinant CPU et GPU. L'hypothèse sera réfutée si les méthodes de traçage et profilage ne permettent pas d'identifier des problèmes de performance ou de suggérer des éléments à optimiser pour l'exécution d'un programme flot de données sur un système hétérogène. Le surcoût introduit par notre travail devra également rester raisonnable, sans quoi l'hypothèse sera également réfutée.

1.4 Plan du mémoire

Ce mémoire suit le plan suivant. Après cette introduction, le chapitre 2 présente une revue critique de la littérature. Elle permet d'informer le lecteur à propos des principaux outils et des technologies abordées dans le mémoire. Le lecteur sera ainsi plus familier avec les différents concepts et thèmes du mémoire et la lecture sera facilitée.

Par la suite, le chapitre 3 décrit la méthodologie utilisée dans le cadre du projet de recherche. Des éléments techniques ainsi que certains points clés de l'implémentation sont présentés. Les configurations matérielle et logicielle sont également décrites, ce qui est bénéfique à la compréhension et pour la reproduction des résultats.

Puis, le chapitre 4 représente la partie principale de ce mémoire. Il prend la forme d'un article de recherche décrivant la méthode développée. Nous y présentons également plusieurs cas d'utilisations afin de démontrer l'efficacité de la solution proposée. Une analyse et une discussion à propos de la performance de notre méthode sont également menées.

Dans le chapitre 5, nous procédons à une analyse critique des résultats du chapitre 4. Nous revenons sur les points clés, notamment le surcoût introduit par notre solution dans différents cas.

Enfin, ce mémoire se termine par une conclusion dans laquelle nous commençons par résumer les travaux de recherche. Nous discutons ensuite des limites de notre solution et évoquons des possibilités de travaux futurs.

CHAPITRE 2 REVUE DE LITTÉRATURE

Ce chapitre constitue un état de l'art des différents domaines en lien avec le travail présenté et pourra faciliter la compréhension du lecteur dans la suite du mémoire. Nous abordons dans un premier temps le modèle flot de données, en présentant notamment des langages et applications utilisant ce genre d'approche. Nous nous intéressons par la suite à TensorFlow (Abadi et al., 2016a,b), une librairie pour l'apprentissage automatique et profond utilisant un graphe de calcul pour représenter les algorithmes et qui constitue la base pour l'implémentation de notre travail. Après cela, nous évoquons les techniques ainsi que les outils existants pour le traçage et le profilage d'applications. Nous visons les processeurs traditionnels dans un premier temps et les processeurs graphiques ensuite. Nous présentons également certains outils permettant la visualisation et l'analyse des traces. Enfin, nous terminons par détailler les travaux existants à propos du profilage et de l'analyse d'applications flots de données.

2.1 Modèle flot de données

Dans cette section, nous rappellerons tout d'abord les principes théoriques des modèles flot de données. Nous présenterons ensuite des langages développés, ainsi que plusieurs méthodes et techniques dans différents domaines qui sont basés sur ce genre d'approches. Enfin, nous aborderons plusieurs bibliothèques logicielles pour l'apprentissage automatique et profond qui utilisent un modèle de calcul flot de données. Nous détaillerons en particulier TensorFlow, pour laquelle l'ensemble du travail présenté dans ce mémoire a été développé.

2.1.1 Principes théoriques

Le modèle flot de données est un paradigme de programmation apparu à la fin des années 1960 qui consiste en un graphe acyclique contenant des nœuds reliés entre eux par des arcs. Chaque nœud représente une unité de calcul ayant une ou plusieurs entrées et sorties. Les arcs correspondent à des liens, le long desquels les données fournies au graphe vont pouvoir circuler. Chaque nœud appliquera une opération spécifique aux données reçues en entrée puis fournira le résultat en sortie. Il est donc possible de définir ce modèle de programmation comme une approche centrée sur les données et leur mouvement, et qui permet de modeler un programme en un groupement de connexions entre des unités de calcul. La figure 2.1 représente un modèle flot de données sous la forme d'un graphe. On peut identifier facilement les 8 nœuds de calcul qui vont opérer sur les données. Ces dernières seront

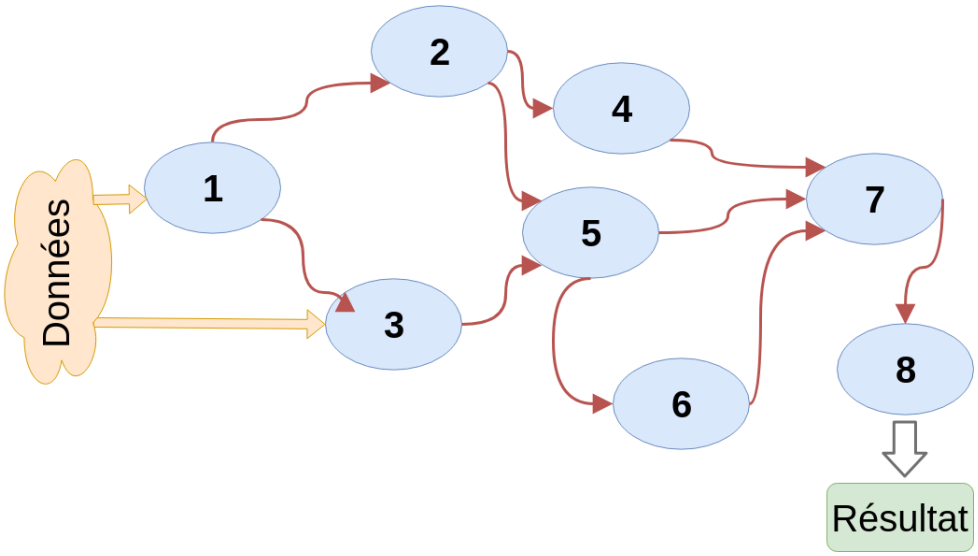


Figure 2.1 Exemple de modèle flot de données. Les données de départ sont fournies en entrée aux nœuds 1 et 3. Elles circulent ensuite le long des arcs et sont traitées par chaque nœud. Le résultat final correspond à la sortie du nœud 8.

fournies au graphe au travers d'un ou de plusieurs nœuds, les numéros 1 et 3 dans cet exemple. Puis, elles circuleront le long des axes jusqu'à atteindre le dernier nœud du graphe, le numéro 8 ici.

Ce modèle de programmation partage des principes similaires avec l'architecture matérielle flot de données. Cette dernière, également connue sous le nom *d'ordinateur cadencé par les données* propose une alternative à l'architecture traditionnelle de von Neumann et sa vision de programmation séquentielle. Dans ce cas, aucun compteur de programme n'est utilisé, mais toute l'exécution repose sur la disponibilité des données en entrée qui déclenchera l'exécution d'une opération. Le fait de séparer le traitement en unité de calcul individuelle, rend ce modèle très bien adapté à la programmation parallèle ou distribuée. Cela constitue une des principales motivations pour son développement et contraste avec l'architecture classique de von Neumann, par essence mal adaptée au principe de parallélisme. Comme de nos jours les traitements en parallèle sont essentiels pour atteindre des performances toujours plus importantes, les approches flots de données sont devenues très populaire.

2.1.2 Langages de type flot de données

Ce modèle étant très répandu, il a inspiré de nombreux langages de programmation et travaux de recherche. Lee (1991) a étudié les propriétés analytiques de ces langages et notamment la cohérence de leur graphe flot de données. Lustre (Caspi et al., 1987; Halbwachs et al., 1991),

par exemple, est un langage déclaratif, synchrone et adoptant le modèle flot de données qui est utilisé pour programmer des systèmes réactifs. Aujourd'hui intégré à l'environnement commercial SCADE, il sert principalement à la conception logicielle de systèmes critiques dans l'aéronautique, le ferroviaire ou encore les centrales nucléaires. Lucid (Wadge and Ashcroft, 1985) est un second exemple de langage de programmation suivant ce genre d'approche. CAL (CAL Actor langage) (Eker and Janneck, 2003) est un autre exemple de langage haut niveau, compilé et destiné à différents types d'architectures hétérogènes (Boutellier and Nyländen, 2017) et (Lin et al., 2016). Il a été développé au sein du projet Ptolemy (Eker et al., 2003), un environnement de développement Java pour la conception, modélisation et simulation hétérogène de systèmes flot de données concurrents. Bhattacharyya et al. (2008) proposent un environnement de compilation et simulation pour CAL, contenant un support pour la génération de code HDL (Verilog ou VHDL) (Janneck et al., 2008) et de code C en vue d'une intégration avec SystemC (Roquier et al., 2008). Par ailleurs, ce langage a été ajouté au nouveau standard MPEG "Reconfigurable Video Coding (RCV)" (ISO/IEC 23001-4 et 23002-4) pour former RVC-CAL. Ce dernier est notamment utilisé pour le traitement de signal (Hentati et al., 2012), la compression (Bezati et al., 2010), la programmation haute performance de GPU (Boutellier and Nylanden, 2015) ainsi qu'en cryptographie. Dans le domaine de la conception matérielle, de nouveaux langages de description ont été développés (Port and Etsion, 2017), tout comme dans celui du Big Data où des approches flot de données innovantes ont vu le jour comme SYRql (Maali et al., 2014) qui vise à traiter des données RDF à très grande échelle.

2.1.3 Méthodes et techniques flot de données

Étant très bien adapté au principe de parallélisme, ce modèle de programmation constitue un sujet de recherche important et très étudié. Le domaine du traitement de signal, d'image et de vidéos l'utilise par exemple régulièrement. Boutellier et al. (2018) proposent un modèle de calcul dynamique basé sur une approche flot de données pour le déploiement d'applications de traitement du signal dans des environnements matériels hétérogènes. Bourrasset et al. (2016a) décrivent une implémentation flot de données d'un détecteur d'objets configurables sur une plateforme constituée de FPGAs. Enfin, Blattner et al. (2017) proposent un ordonnanceur de tâches hybrides dans un graphe en se basant également sur une approche flot de données. D'autres domaines comme les réseaux (Kohler et al., 2000) ou la conception matérielle (Li et al., 2016) bénéficient également des avancés de ces modèles. Plusieurs méthodes et techniques de conception utilisent un modèle unique flot de données afin de synthétiser conjointement les composants matériels et logiciels ainsi que leurs interfaces (Roquier et al., 2011) et (Carlsson et al., 2010). Le niveau d'abstraction élevé apporté par ces approches

est tout particulièrement apprécié dans le cadre d'architectures hétérogènes. Enfin, des techniques inspirées des approches flot de données sont également apparues comme TIDeFlow (Orozco et al., 2016) qui propose l'utilisation d'un graphe orienté pour représenter un programme. Ce dernier offre différents avantages, comme la possibilité d'exprimer nativement des boucles parallèles afin de modéliser un programme avec un meilleur niveau d'abstraction, et l'introduction du pipeline de tâches permettant un gain de performance.

2.1.4 Modèle flot de données appliqué à l'apprentissage automatique

Dans le domaine de l'apprentissage automatique et de la fouille de données, ce type de modèle est également apprécié. En effet, les applications pour l'apprentissage sont en général très exigeantes en termes de calcul et de quantité de données à traiter. Pour cela, de nouvelles architectures matérielles hétérogènes et parallèles ont été mises en place, où les processeurs traditionnels sont supportés par d'autres composants matériels comme les processeurs graphiques ou encore les "processeurs de tenseurs" *Tensor Processing Unit (TPU)* en anglais (Jouppi et al., 2017). Comme expliqué précédemment, les modèles flots de données sont particulièrement adaptés pour obtenir une performance maximale avec ce genre d'architecture hautement parallèle. Orange (Demšar et al., 2013) est un outil libre de sources pour l'apprentissage automatique, la fouille de données ainsi que la visualisation et propose un fonctionnement basé sur des unités de calcul connectées entre elles qui vont traiter continuellement de nouvelles données. Theano (Bergstra et al., 2010; Al-Rfou et al., 2016) est une librairie pour l'apprentissage automatique et profond qui utilise un modèle de calcul flot de données.

TensorFlow (Abadi et al., 2016a,b, 2017) est un autre exemple de bibliothèque logicielle libre de sources pour l'apprentissage machine et profond qui présente les calculs sous la forme d'un graphe direct et acyclique. Tensorboard (Wongsuphasawat et al., 2018), un outil de visualisation, intégré à TensorFlow permet notamment de visualiser le graphe de calcul d'une application. La figure 2.2 présente un exemple de graphe pour un réseau de neurones basique contenant deux couches cachées et une couche de sortie. Cette librairie, développée majoritairement par Google, est extrêmement populaire dans le domaine de l'intelligence artificielle. Elle propose une implémentation complète et très performante du modèle flot de données. De plus, elle est utilisée par un très grand nombre de personnes et reste activement développée. Pour toutes ces raisons, TensorFlow constitue une excellente base pour l'implémentation de nos travaux.

Les nœuds du graphe correspondent à des opérations qui vont traiter les données circulant le long des axes. Dans un contexte d'apprentissage automatique, ces données sont habituel-

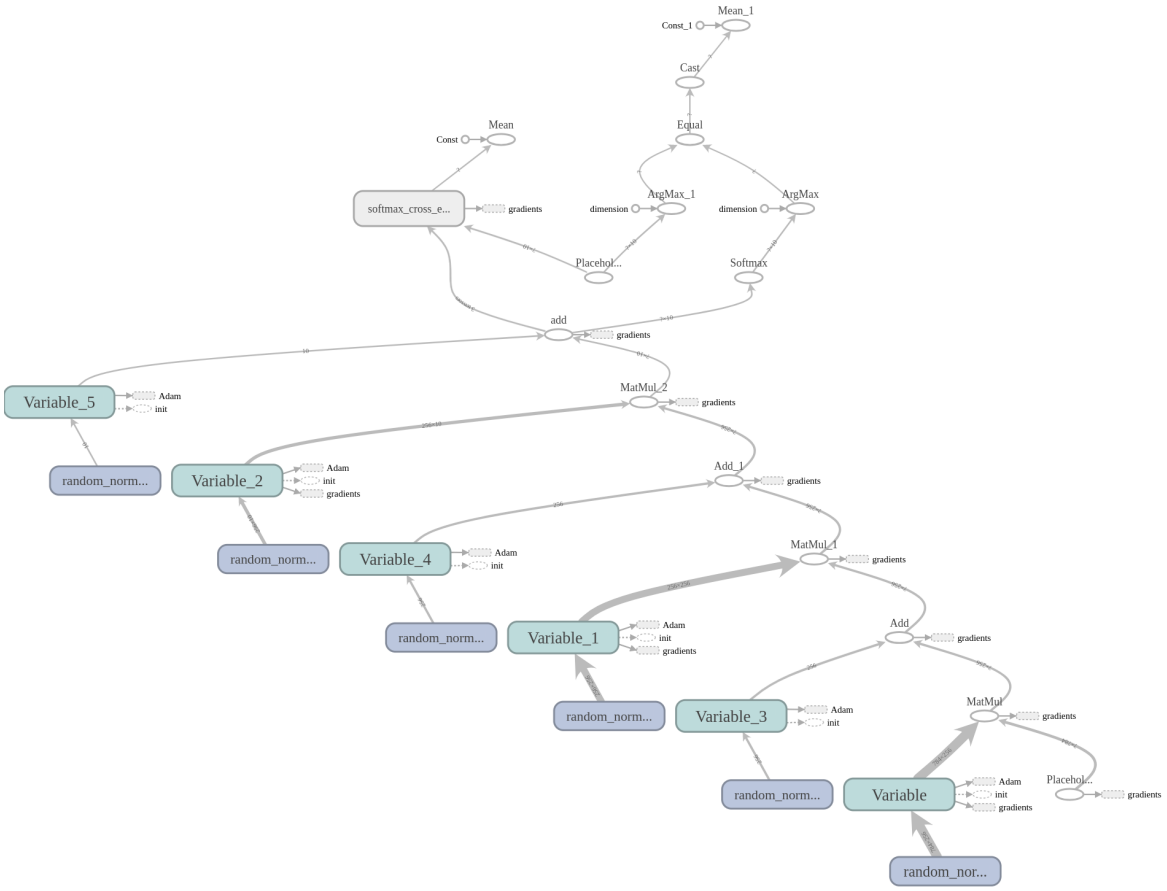


Figure 2.2 Tensorboard : visualisation du graphe d'un réseau de neurones ayant deux couches cachées et une couche de sortie. Une partie annexe du graphe va gérer la propagation arrière et le calcul des gradients mais n'est pas représentée ici.

lement des tableaux multi-dimensionnels appelés *tenseurs* ou *tensors* en anglais. Chaque opération sera associée à une entité matérielle de calcul : processeur central, processeur graphique ou encore processeur de tenseurs. TensorFlow ajoute automatiquement des nœuds au graphe de calcul afin de gérer les transferts de mémoire nécessaires lorsque des nœuds liés ensemble sont associés à différentes entités physiques. Par ailleurs, TensorFlow propose également de réaliser de l'apprentissage de manière distribuée sur plusieurs machines. Dans ce cas, le graphe de calcul peut être partagé sur plusieurs ordinateurs et chaque nœud sera associé à une entité de calcul sur une des machines utilisées. À nouveau, la librairie ajoutera automatiquement des nœuds nécessaires pour gérer les transferts de mémoire.

Dans le cas de TensorFlow, l'approche flot de données facilite l'exécution sur des plateformes hétérogènes. En effet, il est facile d'envisager une exécution sur plusieurs entités physiques du

fait de la décomposition du programme en plusieurs opérations individuelles. Chaque nœud pourra donc être associé à un composant de calcul physique sous la seule condition que ce dernier soit capable d'effectuer l'opération correspondante. Pour un processeur graphique, un noyau de calcul doit par exemple avoir été implémenté. En général, on cherchera à utiliser le processeur graphique pour la plupart des nœuds exigeants en termes de calcul et effectuant un traitement sur un grand nombre de données. On retrouve notamment la majorité des opérations mathématiques comme les fonctions d'activation (*Softmax*, *Relu*, *Sigmoid*) et celles impliquant des additions ou des multiplications de matrices comme les convolutions 2D.

D'un point de vue du code, une application utilisant TensorFlow contient toujours deux étapes distinctes. Dans un premier temps, l'utilisateur doit définir l'algorithme en combinant différentes opérations disponibles, à partir desquelles le graphe de calcul sera créé. C'est ce dernier qui va permettre l'approche flot de donnée. Dans un second temps, un objet *Session* sera créé et on pourra l'utiliser afin de fournir des données en entrée du graphe, démarrer l'exécution et récupérer un ou plusieurs résultats en sortie. Le graphe défini dans la première étape sera lié à la *Session* et c'est elle qui va gérer l'ensemble de l'exécution. Habituellement, les algorithmes d'apprentissage automatique ou profond, comme les réseaux de neurones ou réseaux de neurones convolutifs, impliqueront un très grand nombre d'exécutions du graphe de calcul, avec à chaque fois de nouvelles données en entrée. L'exécution sera déclenchée par un appel à la méthode *Run* de l'objet *Session*. On fournira à cet appel un ou plusieurs nœuds du graphe, dont on souhaite récupérer les résultats en sortie ainsi que les données d'entrées, généralement sous forme de lots. Les figures 2.3 et 2.4 présentent respectivement un exemple de code effectuant plusieurs opérations sur des matrices ainsi que le graphe correspondant.

Comme on l'a vu, TensorFlow se base sur une approche flot de données contenant de nombreux nœuds qui vont appliquer des opérations sur des données. Par ailleurs, cette bibliothèque vise à être utilisée dans un contexte d'architectures hétérogènes avec différents coprocesseurs pour assister le processeur traditionnel. Pour l'instant, seules les cartes graphiques et les processeurs de tenseur sont supportés mais d'autre unités physique comme les DSPs ou les FPGAs pourraient l'être prochainement. Ainsi, l'association entre les nœuds logiques du graphe et les unités physiques de calcul est importante. Par défaut, TensorFlow effectue ce placement de manière automatique. Toutefois, un utilisateur peut aussi placer manuellement certains nœuds sur des entités physiques. Il sera alors essentiel de s'assurer que la performance de l'exécution soit très bonne, voire optimale. De très nombreux éléments devront être considérés comme les transferts de mémoire, le parallélisme ou encore l'utilisation adéquate des différentes unités de calcul. Aucun outil ne le permet actuellement et c'est un des manques que nous chercherons à combler dans ce projet de recherche.

```
# ...
# Génération de données aléatoire pour fournir au graphe
A_values = np.random.randint(10, size=(10,10))
B_values = np.random.randint(10, size=(10,10))
C_values = np.random.randint(10, size=(10,10))

# déclaration des placeholders, qui vont accueillir les données d'entrées
A = tf.placeholder(shape=(10,10), dtype=tf.float32)
B = tf.placeholder(shape=(10,10), dtype=tf.float32)
C = tf.placeholder(shape=(10,10), dtype=tf.float32)

# déclaration d'une variable
W = tf.Variable(tf.random_uniform([10,1], -1, 1))

# construction du programme sous forme d'un graphe de calcul
tmp = tf.add(tf.matmul(A, B), C)
result = tf.matmul(tf.reduce_sum(tmp, 0, keepdims=True), W)

# Création d'une session et lancement de l'exécution
with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    res = sess.run(result, feed_dict={A: A_values, B: B_values, C: C_values})
    print("result:", res)
```

Figure 2.3 Exemple de programme TensorFlow

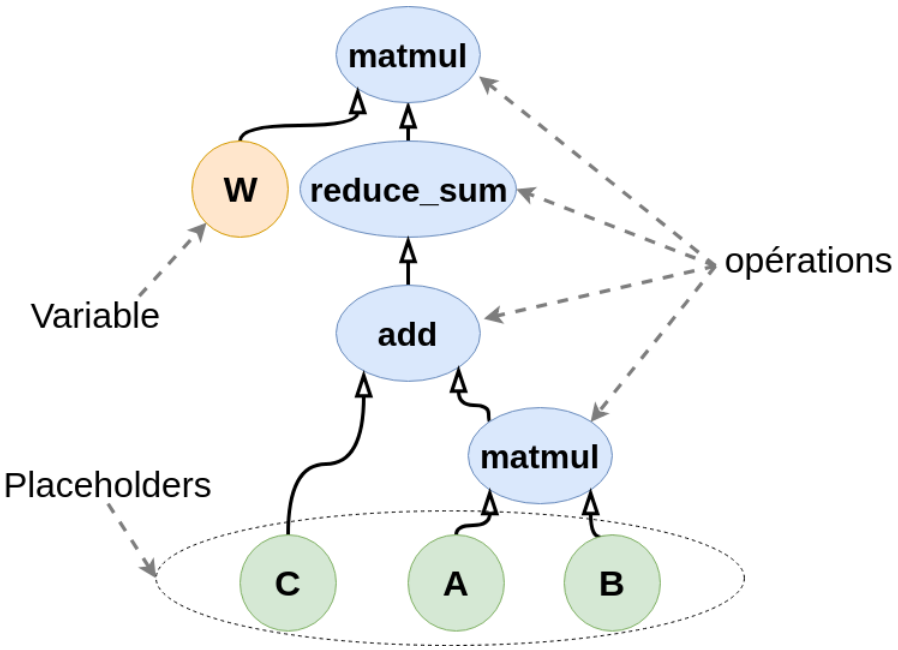


Figure 2.4 Graphe de calcul correspondant au code précédent

2.2 Architectures hétérogènes et processeurs graphiques

Dans cette section, nous allons tout d'abord justifier l'apparition des architectures hétérogènes. Ensuite nous nous intéresserons aux processeurs graphiques en présentant un historique ainsi que quelques cas d'utilisation de ces unités de calcul. Après cela, nous aborderons les architectures existantes et les suites logicielles disponibles pour les programmer.

2.2.1 Architectures hétérogènes

Entre les années 1980 et 2000, l'augmentation des performances des ordinateurs pouvait s'expliquer par loi de Moore (Moore, 1998). Ce dernier a observé en 1965, que la complexité des semi-conducteurs d'entrée de gamme doublait tous les deux ans à coût constant et avait prédit que cette tendance allait durer. Il avait également réévalué sa prédiction quelques années après, en expliquant qu'il s'agissait du nombre de transistors des microprocesseurs qui doublait tous les deux ans. Cependant, depuis quelques années, cette loi se vérifie de plus en plus difficilement. En effet, des limites physiques ont été atteintes, notamment en termes de finesse de gravure et de dissipation de chaleur. Le nombre de transistors sur les microprocesseurs ainsi que leur fréquence ne pourront pas augmenter indéfiniment. Par conséquent, des approches comme le calcul en parallèle sont privilégiées afin de maintenir une augmentation de la puissance de calcul.

La loi d'Amdahl (Amdahl, 1967), énoncée en 1967, définit l'accélération possible en introduisant du calcul en parallèle.

$$\frac{Ts}{Tp} = \frac{1}{1 - p + \frac{p}{n}} \tag{2.1}$$

avec :

- Ts : temps d'exécution d'un programme séquentiel
- Tp : temps d'exécution d'un programme parallélisé
- p : pourcentage de la section parallèle du programme (0<=p<=1)
- n : nombre d'éléments de calcul considérés pour l'exécution de la section parallèle du programme

Ce principe de traitement parallèle correspond à une réelle opportunité d'augmentation de la puissance de calcul dans les ordinateurs actuels. Les processeurs traditionnels sont capables d'effectuer du calcul en parallèle sur plusieurs cœurs mais cela reste tout de même assez limité. En effet, ces unités de calcul sont optimisées pour proposer une latence minimale, mais restent relativement inefficaces pour traiter plusieurs données simultanément. Au contraire,

d'autres entités de calcul comme les processeurs graphiques proposent quant à eux une architecture hautement parallèle. Ils visent à offrir un débit maximal et leur nombreux cœurs leur permettent de traiter un grand nombre de données en même temps. En combinant ces deux types d'unités de calcul, des architectures hétérogènes peuvent être créées afin d'offrir un traitement optimal ayant à la fois les avantages des processeurs traditionnels et ceux des processeurs graphiques.

2.2.2 Historique

Les processeurs graphiques sont des composants matériels apparus dans les années 1970. Ils ont été développés à l'origine pour s'occuper de la partie graphique des applications 2D et 3D, avec notamment l'affichage à l'écran (Singer, 2013). Cela permettait aux processeurs traditionnels d'être libérés d'une partie du travail. Par la suite, les capacités de ces processeurs ont continuellement augmenté avec notamment la popularité croissante des jeux vidéos. Ces derniers avaient des besoins toujours plus importants pour afficher plus de pixels avec le plus de couleurs possibles. À l'époque, OpenGL, une librairie multi-plateforme, ainsi que Direct3D pour les environnements Windows, constituaient les deux solutions majeures pour programmer un processeur graphique.

À partir des années 2000, une nouvelle tendance est apparue, la programmation de cartes graphiques pour des besoins généraux, *GPGPU* en anglais(Owens et al., 2008). Il s'agit d'utiliser les processeurs graphiques pour des tâches sans lien direct avec l'affichage. De nombreuses opérations mathématiques sur des matrices, à possiblement plusieurs dimensions, peuvent par exemple être accélérées à l'aide de ce matériel. Les multiplications de matrices ont justement été l'une des premières applications non graphiques a être implémentées sur un GPU (Larsen and McAllister, 2001) et, quelques années plus tard, la factorisation LU a été l'un des premiers algorithmes mathématiques pour lesquels une implémentation sur processeur graphique était plus performante qu'une version optimisée exécutée sur un processeur classique (Galoppo et al., 2005).

Une des limites de cette nouvelle tendance était la difficulté de programmation de ces composants en comparaison avec les processeurs traditionnels. L'introduction en 2007 de *CUDA* (Compute Unified Device Architecture) par Nvidia constitue un tournant dans le domaine GPGPU (Nvi, 2007, 2018) en rendant la programmation de processeurs graphiques bien plus aisée. Au fur et à mesure des versions de CUDA, cette dernière est devenue plus populaire et envisagée par de plus en plus de programmeurs. Par ailleurs, l'utilisation de ce type de matériel a commencé à être totalement justifiée puisque les implémentations de divers algorithmes commençaient à surpasser les performances des versions optimisées pour un processeur tra-

ditionnel. D'autres librairies ont ensuite vu le jour et seront présentées plus tard.

Aujourd'hui, de très nombreux domaines bénéficient de l'utilisation des GPUs. Les jeux vidéos, tout d'abord, avec le rendu graphique en trois dimensions, constituent un premier exemple. Ces applications requièrent la génération et l'affichage d'un très grand nombre de polygones avec une résolution et taille d'écran toujours plus grande, ce qui nécessite une importante puissance de calcul. L'affichage des textures, les déplacements et rotations de la caméra sont également exigeants. On retrouve ensuite le domaine du multimédia. En effet, les algorithmes de traitement d'images sont particulièrement adaptés pour une utilisation sur GPU, puisqu'il s'agit généralement d'effectuer une même opération sur l'ensemble des pixels d'une image. Il est évidemment plus performant d'appliquer ce traitement en parallèle sur plusieurs pixels simultanément. L'encodage et décodage vidéo sont également propices à une accélération par processeurs graphiques. Les GPUs sont aussi populaires dans les domaines de l'informatique embarquée ou mobile comme avec les téléphones intelligents. En effet, pour certaines tâches, l'utilisation d'un processeur graphique plutôt qu'un processeur traditionnel pourra aboutir à une économie d'énergie, ce qui peut être très bénéfique dans un contexte d'appareils alimentés par batteries. De manière générale, de très nombreux domaines scientifiques utilisent les cartes graphiques afin d'accélérer les calculs. La bio-informatique et notamment le traitement des séquences ADN (Mahmoudi et al., 2011), la thermodynamique (Goddeke et al., 2009) ou encore la simulation d'ondes sismiques (Komatitsch et al., 2010) constituent quelques exemples. Pour le calcul scientifique, des logiciels majeurs comme Wolfram Mathematica (Wol, 2018) et Matlab (Mat, 2015) contiennent un support pour les processeurs graphiques. Enfin, plus récemment, l'intelligence artificielle a connu un intérêt grandissant avec notamment l'apprentissage profond. Ce dernier implique généralement de nombreuses opérations d'algèbre linéaire comme les multiplications de matrices appliquées sur un très grand nombre de données. Il est donc avantageux d'utiliser un GPU pour ces traitements, tel que démontré dans de nombreux travaux de recherche (Lawrence et al., 2017) et (Schlegel, 2015).

2.2.3 Architectures

Aucune architecture générique n'existe pour les processeurs graphiques, car cela dépend fortement du constructeur, mais aussi de l'utilisation finale souhaitée. Par exemple, un GPU destiné à l'affichage proposera une architecture probablement différente d'un modèle orienté pour le calcul scientifique. Au niveau des constructeurs, on retrouve trois acteurs majeurs : Nvidia, AMD et Intel.

Les deux premiers dominent très largement le marché des cartes graphiques dédiées. Intel,

quant à eux, proposent majoritairement des processeurs intégrés. Il s'agit de solutions graphiques incorporées directement sur la carte mère de l'ordinateur et connues sous le nom de *Processeur Graphique Intégré* ou *Integrated Graphic Processor* (IGP) en anglais. Deux options existent dans ce cas. Le processeur graphique intégré peut être associé au CPU comme dans les modèles *Intel HD Graphics* ou *Accelerated Processing Unit* (APU) d'AMD, ou alors être physiquement indépendant du CPU. Les GPUs intégrés à la carte mère apportent des avantages en termes de coût ainsi que de consommation d'énergie en comparaison d'un GPU dédié. Ils restent cependant moins puissants et mieux adaptés à des tâches moins exigeantes. Les processeurs graphiques dédiés offrent au contraire une plus grande puissance de calcul, mais consomment plus d'énergie et demeurent plus onéreux. Ils sont donc mieux adaptés pour des usages intensifs comme les jeux vidéos ou le calcul scientifique.

Du point de vue du fonctionnement, les processeurs graphiques introduisent du parallélisme de données, en effectuant des opérations de manière simultanée sur un ensemble d'éléments organisés en une structure commune, comme un tableau par exemple.

Si l'on s'intéresse au matériel Nvidia, chaque processeur graphique est composé d'un ou plusieurs *Streaming Multiprocessors (SMs)* qui possèdent chacun différentes ressources : des cœurs de calcul appelés *CUDA cores*, des registres ainsi que des unités mémoires spécialisées. On retrouve par exemple, 80 SMs sur un GPU Tesla V100 ou 56 pour le Tesla P100.

AMD, quant à eux proposent une architecture *Graphic Core Next (GCN)* destinée à des tâches de calcul et à une utilisation dans le cadre d'architectures hétérogènes. Cette dernière comprend quelques similitudes avec les architectures de Nvidia. Les processeurs graphiques sont divisés en un certain nombre de *Compute Units (CUs)* qui contiennent chacune quatre unités *SIMD*. Ces dernières permettent, comme leur nom l'indique, de traiter plusieurs données simultanément.

Quelque soit le constructeur, nous remarquons que les architectures deviennent de plus en plus complexes avec des cartes graphiques pouvant atteindre 4096 cœurs de calculs. D'autres innovations comme les files en mode utilisateur ainsi que la mémoire virtuelle partagée commencent à apparaître. La première permet aux applications de communiquer directement avec la carte graphique sans avoir à passer par le système d'exploitation. Cela réduit le nombre d'appels système et donc le surcoût associé au lancement d'une commande sur le GPU. Il pourrait alors être efficace de déléguer plus de travail au processeur graphique, comme des commandes courtes qui autrement n'auraient pas été suffisamment longues pour compenser le surcoût associé au lancement sur la carte graphique. La mémoire virtuelle partagée, quant à elle, permet d'éviter les copies de mémoire entre la mémoire RAM et la mémoire de la carte graphique. Un espace d'adressage virtuel et unique sera partagé entre les deux mémoires,

ce qui facilitera notamment l'utilisation de pointeurs dans certaines structures de données. Par ailleurs, cette avancée permet également de réduire la latence de soumission d'un calcul au processeur graphique. Toutes ces avancées rendent les interactions entre les composants encore plus intenses et l'analyse de performance devient de plus en plus critique.

2.2.4 Librairies pour le calcul parallèle sur GPU

Dans cette sous-section, nous allons présenter différentes bibliothèques logicielles permettant de programmer un processeur graphique. Nous nous concentrerons sur les solutions pour le traitement général et non graphique.

CUDA

CUDA (Compute Unified Device Architecture) (Kirk, 2007) est une plateforme ainsi qu'une interface de programmation développée par Nvidia depuis 2007. Son apparition a marqué un tournant pour le calcul générique sur processeurs graphiques. CUDA n'est disponible que pour le matériel Nvidia et n'est pas libre de sources. Cette technologie représente néanmoins une solution de référence pour la programmation de cartes graphiques, en proposant d'excellentes performances, de nombreuses optimisations ainsi qu'une certaine facilité d'utilisation. Conçue pour le support de plusieurs langages tels que le C++, C et Fortran, elle offre notamment le principe de "source unique" qui permet d'incorporer directement les noyaux de calcul dans le code source de l'application destiné au CPU.

Du point de vue du fonctionnement, CUDA utilise des queues logicielles appelées *Stream* qui vont permettre de soumettre des tâches au processeur graphique. On retrouve principalement deux types de tâches :

- Les noyaux de calcul, qui correspondent à des fonctions que le processeur graphique exécutera de manière parallèle sur un ensemble de données.
- Les copies de données multidirectionnelles entre les unités physiques : de la mémoire CPU vers la mémoire GPU ou de la mémoire GPU vers la mémoire CPU

Ces opérations seront, en général, exécutées de manière asynchrone puisque l'utilisateur va contrôler le fait de mettre une opération dans la file d'attente, mais c'est ensuite le processeur graphique lui-même qui s'occupera de l'ordonnancement et exécutera l'opération une fois prêt. Il est possible d'introduire des opérations synchrones, mais cela bloquera le processeur central en attendant que l'opération effectuée sur le processeur graphique soit terminée. Bien moins efficace, ce type d'opérations n'est utilisé que dans certains cas précis.

Les opérations présentes dans un *Stream*, sont toujours exécutées dans le même ordre que

leur insertion. Afin d'ajouter un mécanisme de concurrence, il est donc intéressant d'utiliser plusieurs *Streams*. C'est le cas pour les copies de mémoire qui peuvent être réalisées de manière totalement parallèle avec des noyaux de calcul, puisque les composants matériels impliqués sont différents. Cela permettra principalement de masquer la latence introduite par les transferts de mémoire.

Au niveau de l'exécution des noyaux de calcul, CUDA utilise une hiérarchie particulière. On y retrouve des *Threads* regroupés en blocs appelés *Thread blocks* qui eux même sont groupés dans une grille ou *grid* en anglais.

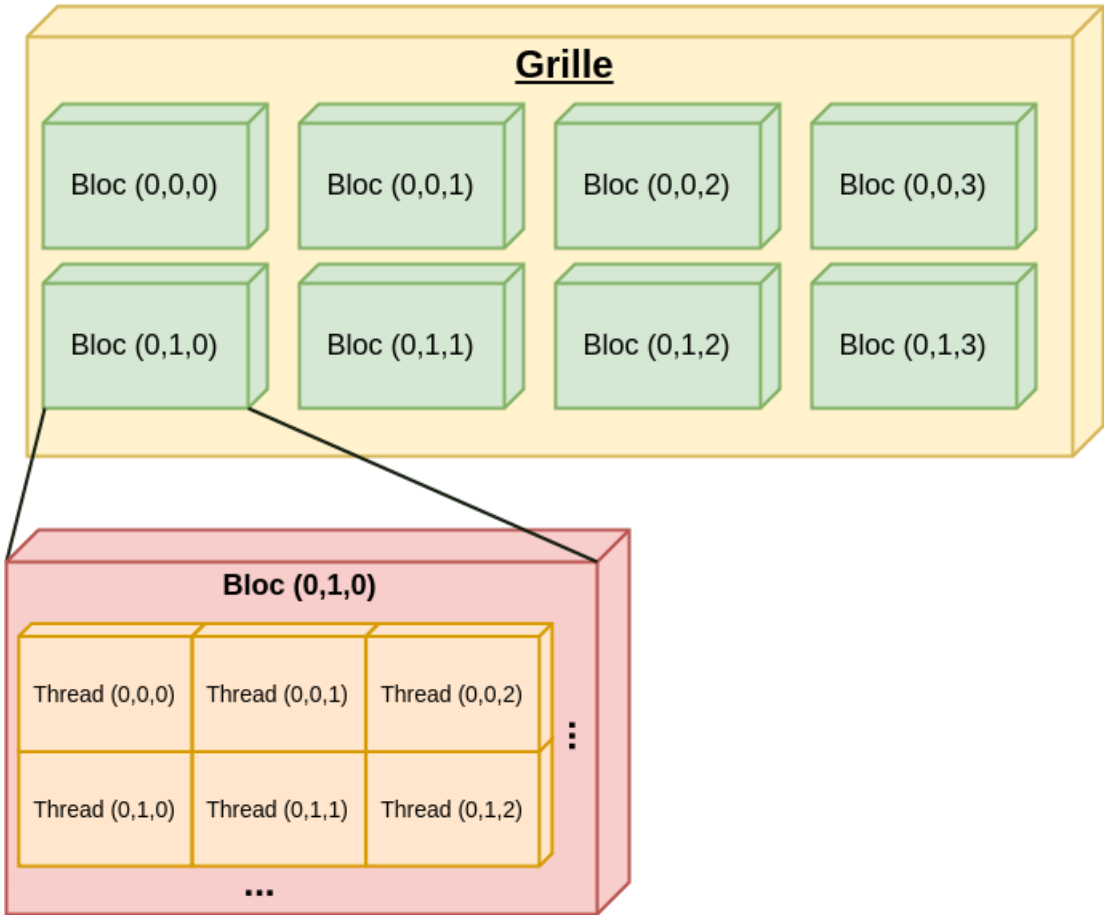


Figure 2.5 Modèle d'exécution de CUDA. La grille et les différents blocs peuvent avoir 1, 2 ou 3 dimensions.

Les blocs ainsi que la grille sont composés de trois dimensions. Au niveau matériel, chaque bloc sera exécuté sur un unique *SM* mais chaque *SM* peut néanmoins exécuter différents blocs. Les *Streaming Multiprocessors* vont toujours créer, gérer, ordonnancer et exécuter les *threads* sous forme de groupes. Ces derniers seront appelés *Warp* et peuvent constituer un

sujet important lors d'une phase d'optimisation ou d'analyse de la performance des noyaux de calcul. En effet, une utilisation appropriée peut permettre de réduire la latence d'exécution. De plus, le concept de *Warp* est utilisé pour définir le taux d'occupation du processeur graphique, ou *occupancy* en anglais. Il se calcule par le rapport entre le nombre de *Warp* actifs et le nombre maximal de *Warp* pouvant être actifs. Ce quotient dépend notamment de la taille des blocs choisie, de l'utilisation de ressources limitées comme les registres, mais aussi de certaines limites matérielles. Bien qu'une valeur de 100% ne soit pas nécessaire, atteindre un haut taux d'occupation est souhaité afin d'utiliser le matériel efficacement.

Un autre élément essentiel lors de la programmation d'une carte graphique concerne la gestion de la mémoire. Comme le montre la figure 2.6, CUDA propose différents types de mémoire

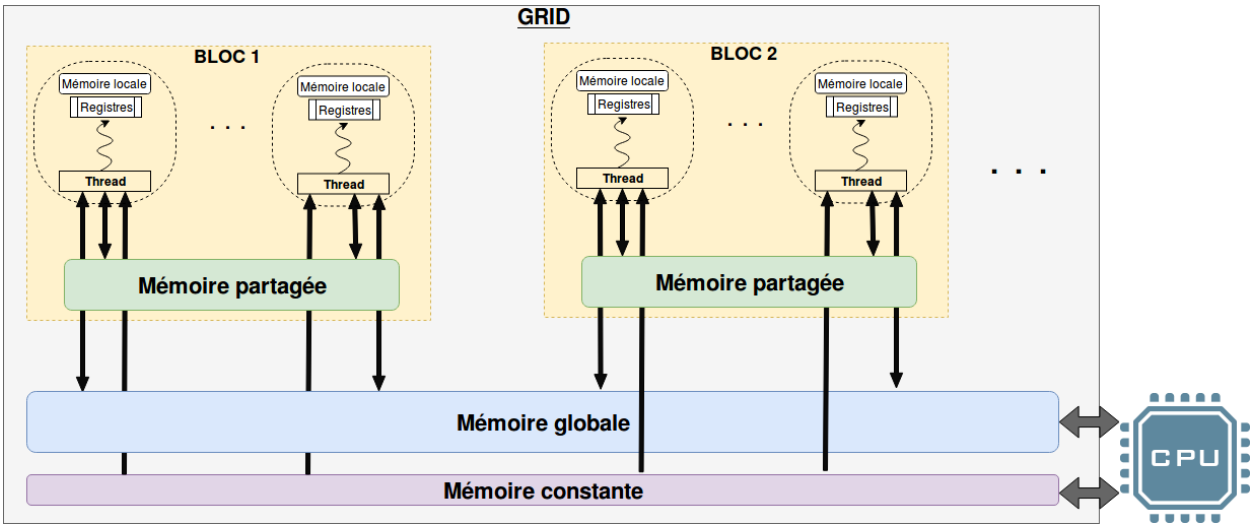


Figure 2.6 Modèle de mémoire utilisé par CUDA

qui correspondent également à des composants distincts au niveau matériel. Chaque mémoire comporte des avantages et inconvénients et une utilisation adéquate et réfléchie est nécessaire afin d'atteindre des performances optimales. En voici une rapide description :

— Mémoire globale : son champ de visibilité est grand puisque tous les *threads* ainsi que le processeur central y auront accès. Les données y persisteront entre les différents noyaux de calcul au sein d'une même application. L'allocation et la déallocation de mémoire sont gérées manuellement par le programmeur avec des appels explicites à des fonctions comme `CudaMalloc` et `CudaFree`. Il s'agit de la mémoire ayant la plus grosse capacité, mais au détriment d'une vitesse relativement faible.

— Mémoire locale ou par *thread* : sa portée ainsi que sa persistance sont limitées au *thread* d'exécution. Physiquement, cette mémoire peut être stockée dans deux com-

posants. Tout d'abord dans des registres qui offrent les meilleures performances en termes de vitesse ou dans une mémoire locale, plus lente, quand la taille des registres est insuffisante.

— Mémoire partagée : cette mémoire offre une visibilité pour tous les *threads* d'un même bloc et une latence plus faible que la mémoire globale. Par ailleurs, sa taille est plus importante que la mémoire locale. Souvent, une utilisation efficace de cette mémoire constituera un point clé pour atteindre une performance maximale.

— Mémoire constante : il s'agit d'un type de mémoire spéciale, de taille fixe et étant disponible uniquement en lecture pour les noyaux de calcul.

2.2.5 OpenCL

OpenCL (Stone et al., 2010; Tompson and Schlachter, 2012), est une interface de programmation libre de sources et multi-plateforme. Il s'agit d'une spécification développée depuis 2009 par le groupe Khronos, un consortium industriel à but non lucratif. OpenCL vise à être utilisé dans le cas d'architectures hétérogènes combinant différentes entités de calcul comme les CPUs, GPUs, DSPs ou encore FPGAs. Il s'agit de la principale alternative à CUDA mais nécessite généralement un travail de développement plus important. Plusieurs implémentations libres et propriétaires sont disponibles et sont principalement utilisées pour du matériel d'AMD ou Intel. En effet, bien que disponible pour le matériel Nvidia, CUDA reste la solution la plus populaire et performante dans ce cas.

Les principes de fonctionnement sont relativement proches de CUDA mais avec un vocabulaire différent. On parle par exemple de *queue de commandes* plutôt que de *streams*. Le modèle d'exécution se rapproche de celui de CUDA avec des *work items* groupés en *work groups*, eux-même regroupés dans une grille. De même, le principe de *warp* se retrouve sous le terme anglophone de *wavefront* et propose des caractéristiques similaires.

En termes de mémoire, le modèle utilise également de la *mémoire privée* pour chaque *work item*, de la mémoire locale, partagée entre tous les *work items* du même *work group* et une mémoire globale ayant une visibilité totale. Une zone de mémoire constante, limitée à la lecture pour les noyaux de calcul, est également disponible.

En comparaison de CUDA, OpenCL propose une approche à un niveau d'abstraction plus bas et nécessite un travail légèrement plus important pour atteindre le même résultat. Par ailleurs, les noyaux de calcul ne peuvent pas être intégrés directement avec le code source s'exécutant sur le processeur central. Ils doivent être écrits sous forme de chaînes de caractères ou dans des fichiers séparés.

Afin de combler certains manques, une approche de plus haut niveau appelée SYCL a été développée (Keryell et al., 2015; Beattie, 2015; Silva et al., 2017). Il s'agit également d'une spécification du groupe Khronos offrant une solution C++ de plus haut niveau et à fichier source unique, pour programmer efficacement des architectures hétérogènes. Elle permet également de conserver les avantages de compatibilité et de performance d'OpenCL. En termes d'implémentation, il existe principalement trois solutions. Deux d'entre elles sont libres de source : *sycl-gtx* et *triSYCL* (Doumoulakis et al., 2017). La troisième implémentation est gratuite mais à sources fermées et est développée par une entreprise écossaise nommée Codeplay. Un compilateur ainsi qu'une librairie appelée *ComputeCpp* sont disponibles. Actuellement, il s'agit de la solution la plus avancée, la plus complète et offrant les meilleures performances.

2.2.6 HSA

HSA (Kyriazis, 2012) constitue un ensemble de spécifications matérielles gratuites proposées par la fondation HSA. Cette dernière est une organisation à but non lucratif fondée par plusieurs entreprises comme AMD, ARM, Samsung ou encore Texas Instrument dont l'objectif est de développer des architectures facilitant le développement de programmes parallèles dans des environnements hétérogènes combinant différents types d'unités de traitement. Un intérêt tout particulier est également accordé à la consommation d'énergie. HSA s'intègre à un niveau relativement bas et peut constituer une base pour d'autres bibliothèques logicielles comme OpenCL ou OpenMP. AMD propose une implémentation du standard HSA au sein de sa plateforme de calcul haute performance ROCm. Cette dernière offre un environnement libre de sources pour utiliser efficacement des processeurs traditionnels et graphiques ensemble. Une implémentation d'OpenCL basée sur HSA est également rendue disponible par AMD.

2.2.7 HIP

La programmation avec HSA ou OpenCL pouvant être relativement complexe, AMD a également proposé une bibliothèque logicielle avec un niveau d'abstraction plus élevé et plus simple d'utilisation. Il s'agit de HIP (*Heterogeneous-Compute Interface for Portability*) qui s'approche fortement de CUDA mais libre de sources et supportant du matériel de différents constructeurs. Afin de favoriser son utilisation, AMD a également développé un outil appelé *hipify* qui permet porter automatiquement une base de code existante CUDA vers HIP.

2.3 Traçage et profilage

Dans cette dernière section, nous allons présenter les techniques et les outils de traçage et de profilage existants. Nous évoquerons dans un premier temps ceux destinés aux processeurs traditionnels et ensuite ceux fonctionnant avec les processeurs graphiques. Dans un second temps, nous présenterons des outils permettant l'analyse et la visualisation des résultats. Enfin, nous verrons des travaux existants qui s'intéressent au profilage de modèles flot de données.

2.3.1 Traçage et profilage du processeur traditionnel

Le traçage et le profilage sont deux techniques très populaires pour l'analyse de performance d'applications et de systèmes utilisant un processeur central. Très appréciées par les développeurs, elles permettent de détecter et résoudre des problèmes parfois difficiles à diagnostiquer.

Le traçage consiste à enregistrer des événements durant l'exécution d'un programme. Cette technique vise également à introduire un surcoût minimal qui dépendra principalement du nombre d'événements que l'on souhaite collecter. Ces derniers formeront une trace, qui sera par la suite analysée et présentée à l'utilisateur. Le traçage implique souvent une instrumentation du code source de l'application de la part de l'utilisateur. C'est pourquoi cette technique est mieux adaptée aux environnements libres de sources comme Linux. Toutefois, il existe également des solutions pour les environnements à sources fermées comme ETW pour Windows.

Le profilage, quant à lui, correspond à la collecte de certaines métriques pendant l'exécution de l'application et implique généralement un échantillonnage. La plupart des processeurs récents fournissent des compteurs de performance qui donnent un aperçu sur l'activité du CPU. Il s'agit d'une information que l'on peut obtenir avec du profilage. Le suivi de la consommation mémoire de la part d'une application constitue un second exemple.

Nous allons maintenant présenter plusieurs outils populaires permettant d'appliquer ces deux techniques.

Strace

`Strace` (strace project, 2010) est un outil de débogage et de diagnostic destiné aux systèmes Linux. Il permet de monitorer et de présenter les interactions entre les différents processus et le noyau Linux. Il correspond à l'un des premiers outils à utiliser lors d'une analyse de performance en raison de sa simplicité d'utilisation. Par ailleurs, il ne requiert aucune

instrumentation ou recompilation de l'application que l'on souhaite tracer. En fournissant simplement une commande en entrée, il est capable de lister l'ensemble des signaux observés ainsi que les appels système accompagnés de leurs arguments et de leur valeur de retour. Lors de son utilisation, `Strace` affiche généralement un très grand nombre de résultats. Il est donc important d'utiliser son système de filtres, permettant de n'enregistrer que certains appels systèmes. On peut par exemple collecter l'ensemble des appels système *open* causés par la commande *date* :

```
pierre@pierre-All-Series ~> strace -e open date
open("/opt/rocm/lib/tls/x86_64/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
open("/opt/rocm/lib/tls/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
open("/opt/rocm/lib/x86_64/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
open("/opt/rocm/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
open("/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
open("/etc/localtime", O_RDONLY|O_CLOEXEC) = 3
```

L'inconvénient majeur de cet outil concerne sa performance. En effet, son fonctionnement est basé sur l'appel système *ptrace* qui permet de contrôler et analyser un processus, mais introduit un ralentissement important de l'exécution (Gregg, 2014). Par ailleurs, `Strace` reste mieux adapté pour des applications utilisant un unique fil d'exécution. En effet, de nombreux mécanismes de verrouillage seront nécessaires lors d'une utilisation pour des processus à plusieurs fils d'exécution, ce qui réduira encore sa performance (Spear et al., 2012). On peut noter par exemple que la sortie est limitée à un seul fichier. Lors du traçage d'application à plusieurs fils d'exécution, il faudrait donc ajouter des primitives de synchronisation afin de gérer les écritures simultanées des résultats dans ce fichier unique, ce qui réduira la performance de l'outil.

Ftrace

En 2008, Rostedt et Molnar ont développé le traceur `Ftrace` (Rostedt, 2009) . Ce dernier est intégré directement dans le noyau Linux et propose un ensemble de fonctionnalités différentes. On retrouve l'analyse du noyau grâce aux nombreux points de traces existants dans ce dernier. Cela permet de collecter des informations à propos de l'ordonnancement, des interruptions, des opérations d'entrée/sortie, ainsi que des opérations liées au système de fichiers et de la virtualisation. L'instrumentation dynamique est également possible grâce à *kprobes*. Lors de son utilisation, la configuration et le contrôle de `Ftrace` s'effectuent au travers du système de fichiers *Trace File System* (TraceFS). Cela diffère des autres outils et peut être déstabilisant pour l'utilisateur dans un premier temps. Cet outil aurait pu se révéler intéressant pour notre travail, mais il est plus orienté vers un traçage au niveau du noyau du système d'exploitation. Comme nous visons à analyser l'activité des différentes unités de calculs impliquées, une

composante majeure sera centrée sur le traçage au niveau utilisateur. Par conséquent, notre choix s'est porté sur un autre outil.

Perf

`Perf` est un outil pour l'analyse de performance disponible dans le noyau Linux à partir de la version 2.6.31. Cet outil se commande à partir d'un terminal et propose une analyse statistique du système. Il permet la collecte de compteurs de performance matériels tels que le nombre d'instructions exécutées ou de fautes de cache. Une instrumentation du code source est également possible en ajoutant des points de trace à des emplacements stratégiques. Enfin, l'instrumentation dynamique est aussi envisageable par le biais de *kprobes* pour le noyau et *uprobes* pour l'espace utilisateur. Tout comme *Ftrace*, *Perf* est plutôt orienté pour l'analyse au niveau du noyau du système d'exploitation et nous estimons qu'il ne s'agit pas du meilleur choix pour notre travail.

DTrace

`DTrace` (Gregg and Mauro, 2011) est un outil de traçage pour l'analyse de performance et la résolution de problèmes dans le noyau du système d'exploitation, ainsi que dans les applications au niveau utilisateur. À l'origine développé par Sun Microsystems, il visait exclusivement le système d'exploitation Solaris. Une équipe d'Oracle a travaillé sur son adaptation pour le noyau Linux (Gregg, 2014, nov). Son principe de fonctionnement consiste à attacher des scripts à des points de trace dans le noyau, ce qui lui permet d'analyser l'ensemble des processus en exécution sur le système. Il peut dresser un aperçu global du système en présentant de nombreuses informations, telles que la quantité de mémoire utilisée, le temps CPU accordé à un fil d'exécution ou encore les ressources liées au réseau ou au système de fichiers. Une analyse plus détaillée est également possible et fournit des informations comme la liste des processus accédant à un certain fichier ou encore les valeurs des différents appels à certaines fonctions. Malheureusement, cet outil comporte quelques faiblesses au niveau de la performance pour quelques applications. En raison de certains mécanismes utilisés, un surcoût significatif apparaît lors d'analyses d'applications ayant plusieurs fils d'exécution (Desnoyers and Dagenais, 2012).

SystemTap

`SystemTap` (Eigler, 2018) propose un fonctionnement proche de `DTrace`, avec des scripts attachés à des points de traces statiques ou dynamiques dans le noyau Linux ou dans une applica-

tion au niveau utilisateur. Ces derniers sont rédigés dans un langage spécifique à `SystemTap` et sont ensuite compilés en langage natif, ce qui assure une grande rapidité d'exécution. `SystemTap` réutilise USDT (*User Statically Defined Tracing*), un mécanisme de `DTrace` pour l'ajout de points de trace dans des applications de l'espace utilisateur. Enfin, contrairement à des outils comme `Strace`, `SystemTap` permet d'analyser l'activité de tous les processus s'exécutant sur le système. Malgré tout, sa performance peut se montrer problématique lors de l'enregistrement d'un grand volume de données.

LTTng

`LTTng` est un outil libre de sources permettant de tracer à la fois le noyau Linux et les applications au niveau utilisateur. Au contraire d'outils comme `Ftrace` et `Perf`, `LTTng` n'est pas intégré dans le noyau Linux mais son module noyau sera chargé au démarrage des outils de traçage. Il vise à proposer un surcoût de traçage très faible et à le maintenir pour les applications parallèles utilisant plusieurs fils d'exécution. Pour cela, `LTTng` utilise un tampon circulaire de taille fixe, pour chaque cœur du processeur, dans lesquels les informations vont être stockées. Cette approche permet notamment d'éviter l'utilisation de mécanismes de synchronisation comme des verrous pour le fichier de sortie. Chaque cœur pourra ainsi écrire les événements collectés sans se soucier de l'écriture des autres cœurs. Une fois pleins, ces tampons seront sauvegardés directement sur le disque dur. Cependant, cette opération peut se révéler relativement longue et la vitesse d'écriture sur le disque dur peut ne pas être suffisamment rapide. Pour gérer ces cas problématiques, `LTTng` a décidé de ne pas introduire de blocage afin d'attendre la fin des écritures sur le disque, mais au contraire de continuer à collecter les informations, en écrasant possiblement des anciens événements dans le tampon circulaire. La taille de ces tampons étant configurable, l'utilisateur doit veiller à choisir une valeur appropriée.

`LTTng` sauvegardera les événements collectés dans le format binaire et standardisé `Common Trace Format (CTF)` (Desnoyers, 2014). Ce dernier a été développé avec un intérêt tout particulier pour la performance, en particulier la vitesse d'écriture d'événements d'une trace.

Une fonctionnalité clé de `LTTng` correspond à son mécanisme de traçage, très efficace dans le domaine utilisateur. Un programmeur peut aisément créer un traceur en définissant un certain nombre de points de trace, puis instrumenter le code source de son application. Par la suite, des événements correspondants aux points de trace insérés seront collectés lors de l'exécution. Cette technique nécessite de recompiler l'application mais permet de comprendre plus en détails le fonctionnement d'une application, et possiblement de détecter des problèmes de performance.

`LTTng` constitue l'outil de base utilisé dans nos travaux puisqu'il combine une facilité d'utilisation, une bonne flexibilité ainsi que des très bonnes performances. Le surcoût introduit est plus faible qu'une grande majorité des autres outils existants. De plus, la possibilité de tracer conjointement le domaine utilisateur et noyau constitue également un avantage à nos yeux. Enfin, différents travaux ont déjà été menés quant à la visualisation et l'analyse des traces obtenues avec cet outil, ce qui conforte notre choix.

2.3.2 Traçage et profilage du processeur graphique

Notre travail a été réalisé dans le cadre d'architectures hétérogènes. Il est nécessaire de s'intéresser au processeur traditionnel, mais également aux autres unités de calcul, les processeurs graphiques dans notre cas. Comme pour les processeurs traditionnels, le besoin d'outils pour l'analyse de performance et la détection de problème s'est très rapidement fait sentir. En raison d'une architecture particulière et d'un fonctionnement différent, les possibilités sont plus limitées pour les processeurs graphiques.

Les techniques de traçage sont applicables et peuvent fournir deux informations essentielles. Tout d'abord, il est possible d'enregistrer l'ensemble des appels de fonction de la bibliothèque logicielle utilisée pour programmer la carte graphique (CUDA, OpenCL, HSA, ...). Ce procédé permet un suivi de l'activité du GPU et se réalise au niveau du processeur central. Il est également possible de collecter les temps de début et de fin de toutes les opérations exécutées par le processeur graphique. Il peut s'agir de noyaux de calcul, de transferts de mémoire ou encore de barrières de synchronisation. Ce procédé intervient quant à lui directement sur la carte graphique. Cependant, il n'est pas possible actuellement d'ajouter une instrumentation ou de collecter des événements au sein des noyaux de calcul s'exécutant sur les cœurs du GPU.

Les techniques de profilage s'appliquent également pour ce type de processeurs. Elles permettent par exemple de suivre la consommation mémoire ainsi que de récupérer des compteurs matériels à propos de l'exécution des noyaux de calcul.

Nous verrons que la majorité des outils de profilage sont développés par les constructeurs de cartes graphiques, mais que certaines solutions indépendantes du matériel et libres de sources existent également.

Processeurs graphiques Nvidia

Nvidia, l'un des constructeurs majeurs de processeurs graphiques dédiés, propose différents outils pour l'analyse de performance de leur matériel. `Nvprof` est une solution de profilage

pouvant être utilisée en ligne de commande et qui fournira des résumés statistiques sous forme de texte dans la console ou dans un fichier de sortie. `Nvprof` est capable de collecter les durées des opérations effectuées par le processeur graphique comme l'exécution de noyaux de calcul ou le transfert de données entre la mémoire vidéo et la mémoire RAM (voir figure 2.7) . Par ailleurs, de nombreux compteurs de performance peuvent être récupérés. La disponibilité de ces derniers dépend du matériel utilisé. En effet, ils sont collectés pour chaque noyau de calcul exécuté et nécessitent des registres afin de stocker leur valeur avant qu'elles ne soient transférées en mémoire RAM.

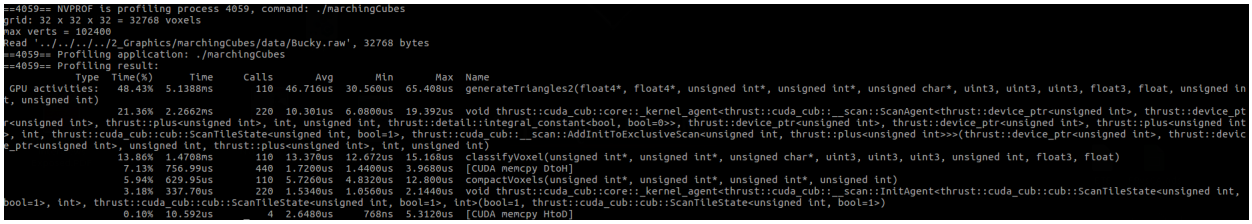


Figure 2.7 Exemple de résultat de profilage avec `nvprof`. On retrouve une ligne pour chaque opération effectuée sur le processeur graphique (noyau de calcul, copie de données). Plusieurs statistiques sont disponibles : le nombre total d'appels, la durée totale, la durée moyenne, les durées maximale et minimale.

Un second outil appelé `Nvidia Visual Profiler` offre une interface graphique pour l'analyse de performance. Il permet de profiler des applications utilisant un processeur graphique et s'appuie sur `nvprof` en arrière-plan. Cet outil peut aussi être simplement utilisé pour la visualisation des résultats de profilage obtenus avec `nvprof`. `Nvidia Visual Profiler` offre de nombreuses analyses pouvant être utilisées individuellement, mais aussi une analyse guidée et complète. Cette dernière facilite et automatise le travail d'optimisation pour un utilisateur et émet des suggestions afin d'améliorer l'application analysée. Le processus d'optimisation proposé par `NSight Visual Profiler` peut être décomposé en deux catégories. On retrouve tout d'abord l'analyse au niveau de l'application qui se concentre sur l'utilisation efficace de la mémoire, la concurrence des commandes envoyées à la carte graphique et le taux d'utilisation de cette dernière (figures 2.8 et 2.9). La seconde analyse s'effectue au niveau des noyaux de calculs. Elle permet d'analyser chaque noyau de calcul exécuté individuellement et identifie des possibilités d'optimisation (figures 2.10, 2.11 et 2.12).

Les résultats du profilage avec `nvprof` ou `NSight Visual Profiler` peuvent être sauvegardés dans un fichier sous un format binaire et propriétaire de Nvidia. Ainsi, l'ensemble des analyses effectuées avec ces outils sont donc destinées à une utilisation au sein de l'environnement `NSight Visual Profiler`. Cela pose clairement un problème de flexibilité, puisqu'il serait difficile d'intégrer de tels outils dans un autre environnement d'analyse. Par ailleurs,

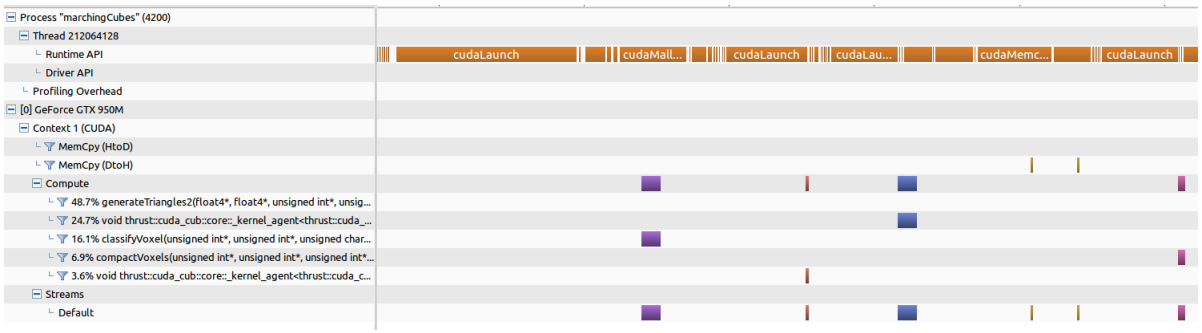


Figure 2.8 Graphique temporel de l'exécution d'une application. Cette visualisation est disponible avec `Nvidia Visual Profiler`. On retrouve plusieurs lignes représentant les appels de fonction à l'API CUDA, les transferts de mémoire, les noyaux de calculs exécutés par le GPU ainsi que les streams (ou files) dans lesquelles les opérations destinées au processeur graphique ont été soumises.

Rank	Description
100	[87 kernel instances] generateTriangles2(float4*, float4*, unsigned int*, unsigned int*, unsigned char*, uint3, uint3, uint3, float3, float, unsigned int, unsigned int)
33	[77 kernel instances] generateTriangles2(float4*, float4*, unsigned int*, unsigned int*, unsigned char*, uint3, uint3, uint3, float3, float, unsigned int, unsigned int)
32	[270 kernel instances] classifyVoxel(unsigned int*, unsigned int*, unsigned char*, uint3, uint3, uint3, unsigned int, float3, float)
32	[279 kernel instances] void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__scan::ScanAgent<thrust::device_ptr<unsigned int>, thrust::device_ptr<unsig
22	[99 kernel instances] void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__scan::ScanAgent<thrust::device_ptr<unsigned int>, thrust::device_ptr<unsigi
14	[104 kernel instances] void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__scan::InitAgent<thrust::cuda_cub::cub::ScanTileState<unsigned int, bool=1>
12	[37 kernel instances] generateTriangles2(float4*, float4*, unsigned int*, unsigned int*, unsigned char*, uint3, uint3, uint3, float3, float, unsigned int, unsigned int)
10	[235 kernel instances] compactVoxels(unsigned int*, unsigned int*, unsigned int*, unsigned int)
8	[149 kernel instances] void thrust::cuda_cub::core::_kernel_agent<thrust::cuda_cub::__scan::ScanAgent<thrust::device_ptr<unsigned int>, thrust::device_ptr<unsig

Figure 2.9 L'analyse automatique de `Nvidia Visual Profiler` propose un classement des noyaux de calcul en fonction de l'importance de leur optimisation. Les critères utilisés sont la durée d'exécution ainsi que le taux d'occupation atteint par chaque noyau. Ainsi, les noyaux les mieux classés sont les plus intéressants à optimiser afin de gagner en performance.

ces outils se concentrent sur l'analyse du processeur graphique, en offrant des informations détaillées, mais d'autres éléments importants manquent. Dans notre cas, le profilage du processeur graphique est important, mais non suffisant, puisque nous visons à collecter des informations à différents niveaux (système d'exploitation, modèle de données, ...) et à propos de toutes les entités physiques utilisées (CPU également). Pour cela, d'autres outils seront nécessaires, mais une combinaison avec `nvprof` ou `NSight Visual Profiler` semble difficile.

Des possibilités de débogage des noyaux de calculs sont également offertes au sein de l'environnement de développement CUDA appelé `Nsight Eclispe Edition`.

Dans notre contexte, une autre bibliothèque de profilage développée par Nvidia semble prometteuse. Il s'agit d'une interface de programmation appelée `CUPTI`, disponible directement au sein de l'environnement CUDA. Cette dernière offre différents modes de fonctionnement pour enregistrer l'ensemble des appels de fonctions CUDA et les durées d'exécution des opé-

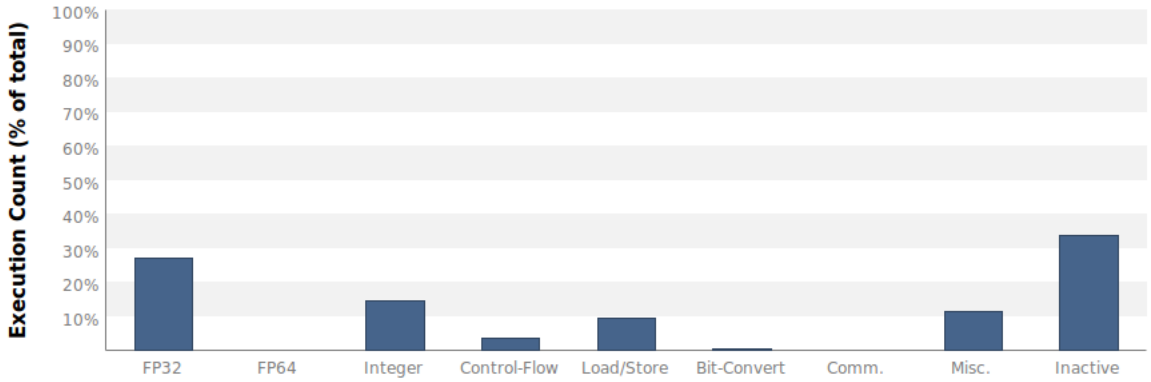


Figure 2.10 Graphique présentant le nombre d'exécution de chaque type d'instruction sous la forme de pourcentage. Cela concerne l'exécution d'un unique noyau de calcul.

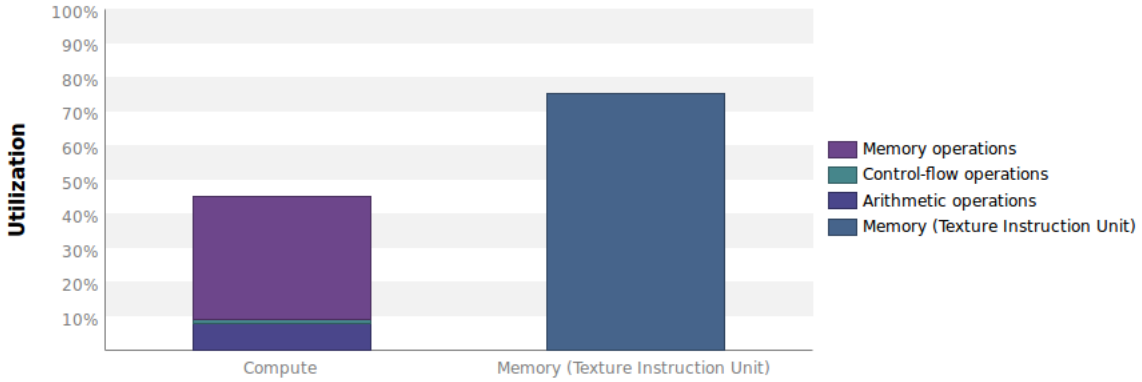
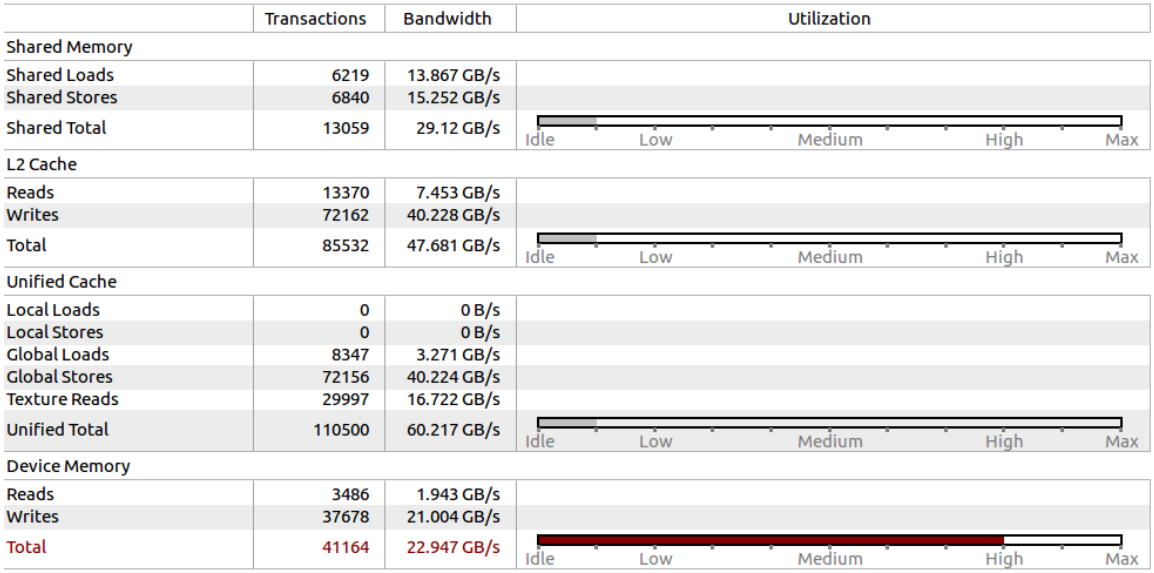


Figure 2.11 Graphique proposé par `Nvidia Visual Profiler` montrant l'utilisation de l'unité de calcul et de mémoire. Il permet d'expliquer à l'utilisateur si l'exécution d'un noyau de calcul est limitée par les calculs ou par la mémoire.

rations asynchrones, mais aussi pour récupérer les compteurs de performance. De nombreuses fonctionnalités sont implémentées à l'aide de fonctions de rappel, ou "callback" en anglais, qui seront appelées à certains moments spécifiques lors de l'exécution d'un noyau de calcul. Cette bibliothèque peut, par exemple, être utilisée pour développer un module d'analyse de l'activité du processeur graphique au sein d'une application. Un travail supplémentaire de programmation est requis, mais cela offre une meilleure flexibilité. Les résultats obtenus ne seront pas liés à l'environnement offert par Nvidia et l'utilisateur pourra choisir comment les traiter et les analyser.

D'autres outils sont en cours de développement par Nvidia et devraient être disponibles dans les prochains mois. Il s'agit d'un ensemble de trois outils de la suite `Nsight` :

— `NSight Systems` : pour analyser l'activité du processeur graphique avec les appels de



	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	6219	13.867 GB/s	
Shared Stores	6840	15.252 GB/s	
Shared Total	13059	29.12 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	13370	7.453 GB/s	
Writes	72162	40.228 GB/s	
Total	85532	47.681 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	8347	3.271 GB/s	
Global Stores	72156	40.224 GB/s	
Texture Reads	29997	16.722 GB/s	
Unified Total	110500	60.217 GB/s	Idle Low Medium High Max
Device Memory			
Reads	3486	1.943 GB/s	
Writes	37678	21.004 GB/s	
Total	41164	22.947 GB/s	Idle Low Medium High Max

Figure 2.12 Information détaillée à propos de l'utilisation des différents composants de mémoire (mémoire partagée, cache L2, cache unifié, mémoire globale).

fonctions CUDA, les exécutions des noyaux de calcul et les transferts de mémoire.

— `NSight Compute` : pour la collection de données bas niveau correspondant à l'exécution d'un noyau de calcul sur le matériel utilisé. Il sera possible d'enregistrer de nombreuses métriques, mais également de comparer plusieurs exécutions d'un même noyau avec des paramètres différents comme la dimension de la grille et des blocs pour la répartition du travail.

— `NSight Graphic` : pour l'analyse d'applications graphiques utilisant des bibliothèques telles que OpenGL.

Le fait d'être développé et maintenu par Nvidia assure une certaine stabilité et une compatibilité avec leur matériel. Malgré tout, une faiblesse apparaît puisque ces outils offrent peu de flexibilité. Le fonctionnement interne n'est pas documenté et le code source n'est pas disponible. Par ailleurs, les résultats collectés sont stockés sous un format binaire et propriétaire, ce qui force l'utilisateur à rester dans cet environnement d'analyse. De plus, les vues graphiques et analyses des résultats sont fixes et ne peuvent pas être configurées ou adaptées par un utilisateur en fonction de ses besoins. Hormis CUPTI, qui offre une certaine liberté, il est donc difficile d'envisager d'intégrer ces outils avec d'autres, ce qui constitue un obstacle pour notre travail.

Processeurs graphiques AMD

AMD propose une plateforme de référence appelée `CodeXL` pour l'analyse de performance d'applications utilisant un GPU. Cet outil libre de sources contient une interface graphique et offre des solutions de profilage mais aussi de débogage. Plusieurs interfaces de programmation pour GPU sont supportées comme OpenCL, HSA ou même OpenGL dans un cadre plus graphique. `CodeXL` correspond à une association de différents outils également libres de sources, qui peuvent aussi être utilisés de manière individuelle. On retrouve par exemple `GPA (GPU Performance API)`, une bibliothèque destinée à la collection de compteurs de performance ou encore `RCP (Radeon Compute Profiler)` pour récupérer les durées des opérations effectuées par le processeur graphique. Au sein de cet environnement, les traces de sortie sont au format texte ATP. Bien que peu efficace dans le cas d'un grand nombre d'événements collectés, ce format possède l'avantage d'être compréhensible par un humain, et donc de pouvoir être réutilisé directement. Les résultats du profilage peuvent ainsi être affichés directement au format texte ou de manière graphique dans `CodeXL`.

La figure 2.16 propose un graphe temporel concernant l'exécution d'une application TensorFlow. On retrouve tout d'abord l'ensemble des appels aux fonctions offertes par l'interface de programmation HSA. Ces fonctions sont exécutées sur le processeur central et les résultats sont regroupés par fil d'exécution. On distingue ensuite l'ensemble des noyaux de calcul exécutés par l'application et groupés en fonction de la file logicielle de la carte graphique dans laquelle ils ont été soumis. De même, les transferts de mémoire entre les différents composants sont visibles. La vue temporelle est complétée par un tableau qui présente en détail l'ensemble des appels de fonction à l'API HSA. On y retrouve les arguments utilisés, la valeur de retour ainsi que le temps CPU nécessaire pour exécuter la fonction.

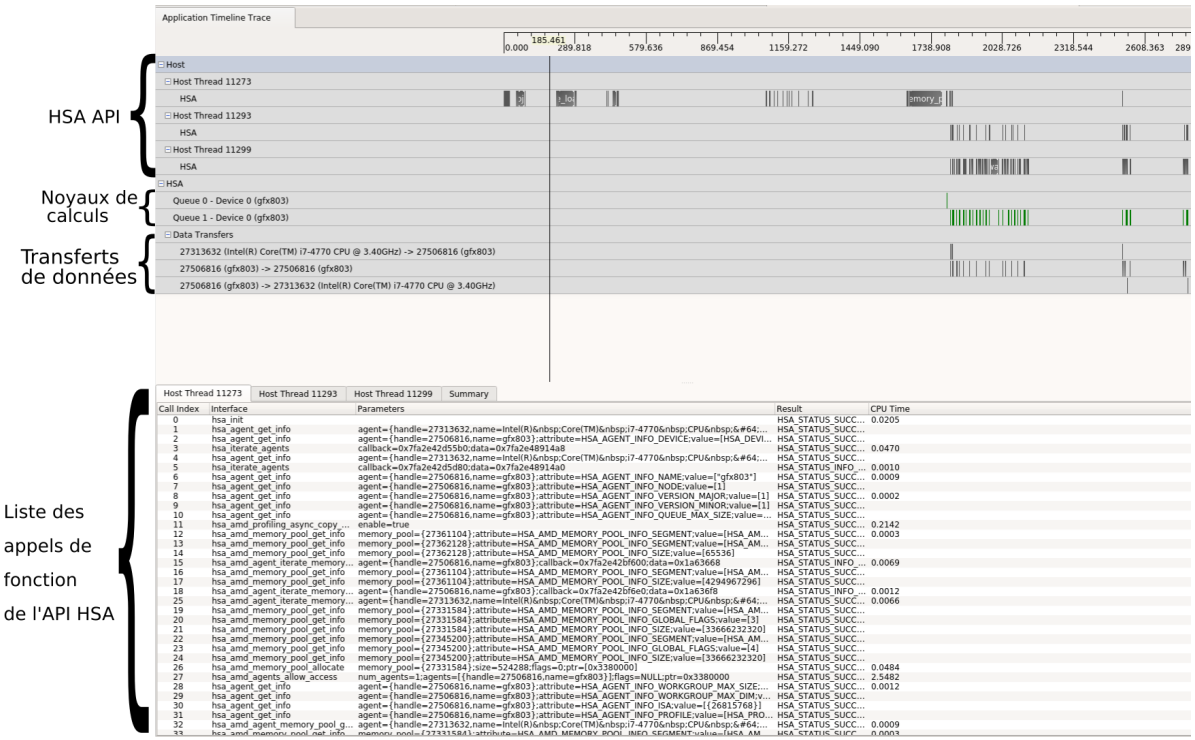


Figure 2.13 Vue temporelle CodeXL. On peut suivre l'ensemble des appels de fonction de l'API HSA, les noyaux de calculs exécutés ainsi que les transferts de données entre la mémoire centrale et celle de la carte graphique.

AMD propose également une bibliothèque d'instrumentation appelée `AMD markers` qui permet à l'utilisateur d'analyser plus précisément le comportement de son application. Lors du profilage d'une application instrumentée, un événement sera généré à chaque marqueur rencontré, et ces derniers seront directement intégrés avec les résultats du profilage. Au niveau de la visualisation, de nouvelles lignes seront ajoutées à la vue temporelle afin de représenter ces événements, comme le montre la figure 2.14.

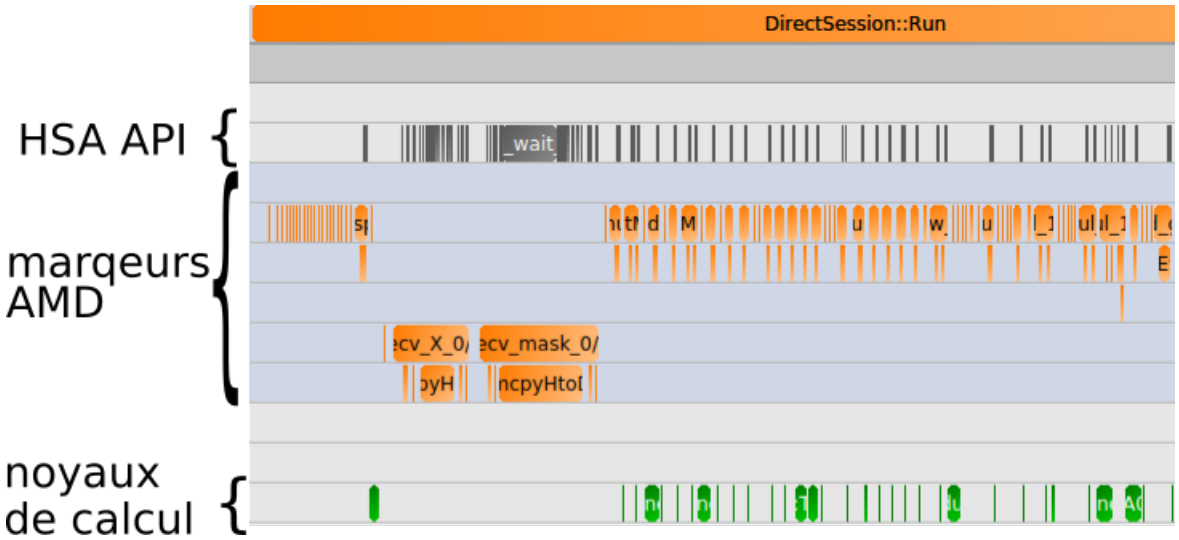


Figure 2.14 Vue temporelle du profilage d'une application ayant été instrumentée avec les marqueurs AMD. L'ensemble des événements liés à l'instrumentation statique sont présentés en orange.

Les compteurs de performance peuvent également être collectés et sont visibles sous la forme d'un tableau dans CodeXL, comme le montre la figure 2.15. Ils représentent différentes métriques pouvant donner des informations clefs à l'utilisateur quant à l'exécution d'un noyau de calcul sur la carte graphique. Chaque ligne correspond à l'exécution d'un noyau de calcul de l'application. Les résultats sont affichés selon l'ordre chronologique des exécutions des noyaux de calcul. Cette présentation des résultats comprend malgré tout une limite dans le cas d'un grand nombre de noyaux de calcul. En effet, il est compliqué de relier chaque noyau avec leur exécution au niveau de la vue temporelle. Ainsi, il est difficile pour un utilisateur de comprendre quelle partie d'une application a soumis un noyau au GPU, dont l'exécution n'est pas optimale au vu des valeurs de certains compteurs de performance.

	ThreadID	GlobalWorkSize	WorkGroupSize	VGPRs	SGPRs	Wavefronts	VALUInsts	SALUInsts	FetchSize	WriteSize	CacheHit (%)
void ihipMemsetKernel<unsigned	303	{ 512 1 1}	{ 256 1 1}	7	13	8	25.50	6.12	0.56	0	16.13
tensorflow::functor::FillPhiloxRandom<E	329	{ 163840 1 1}	{ 256 1 1}	42	84	2560	692.04	105.06	7.56	0	88.41
tensorflow::functor::FillPhiloxRandom<E	329	{ 163840 1 1}	{ 256 1 1}	42	84	2560	743.03	107.81	7.56	0	28.67
tensorflow::functor::FillPhiloxRandom<E	329	{ 163840 1 1}	{ 256 1 1}	42	84	2560	3954.66	281.41	7.56	11403.47	4.29
tensorflow::functor::FillPhiloxRandom<E	329	{ 163840 1 1}	{ 256 1 1}	42	84	2560	701.41	105.56	7.56	0	63.38
tensorflow::functor::FillPhiloxRandom<E	329	{ 163840 1 1}	{ 256 1 1}	42	84	2560	691.26	105.01	7.56	0	91.39
tensorflow::functor::FillPhiloxRandom<E	329	{ 163840 1 1}	{ 256 1 1}	42	84	2560	691.26	105.01	7.56	0	91.26
tensorflow::functor::FillPhiloxRandom<E	329	{ 163840 1 1}	{ 256 1 1}	42	84	2560	692.04	105.06	7.56	0	87.58
tensorflow::functor::FillPhiloxRandom<E	329	{ 163840 1 1}	{ 256 1 1}	42	84	2560	691.27	105.01	7.69	0	91.33
Eigen::internal::TensorExecutor<Eigen::	329	{ 1024 1 1}	{ 512 1 1}	12	16	16	37.25	13	4.06	0	5.74
Eigen::internal::TensorExecutor<Eigen::	329	{ 51200 1 1}	{ 512 1 1}	12	16	800	37.25	13	203.56	0	2.11
Eigen::internal::TensorExecutor<Eigen::	329	{ 163840 1 1}	{ 512 1 1}	12	16	2560	112.40	23.80	12547.62	11836.69	0.07
Eigen::internal::TensorExecutor<Eigen::	329	{ 10240 1 1}	{ 512 1 1}	12	16	160	37.25	13	42.06	0	4.09
Eigen::internal::TensorExecutor<Eigen::	329	{ 512 1 1}	{ 512 1 1}	12	16	8	35.12	12.50	0.88	0	0
Eigen::internal::TensorExecutor<Eigen::	329	{ 512 1 1}	{ 512 1 1}	12	16	8	35.12	12.50	1	0	0
Eigen::internal::TensorExecutor<Eigen::	329	{ 1024 1 1}	{ 512 1 1}	12	16	16	37.25	13	4.94	0	4.67
Eigen::internal::TensorExecutor<Eigen::	329	{ 512 1 1}	{ 512 1 1}	12	16	8	36.75	12.88	0.88	0	6.25
Eigen::internal::TensorExecutor<Eigen::	329	{ 1024 1 1}	{ 512 1 1}	12	16	16	37.25	13	4.06	0	5.74
Eigen::internal::TensorExecutor<Eigen::	329	{ 51200 1 1}	{ 512 1 1}	12	16	800	37.25	13	203.56	0	2.11
Eigen::internal::TensorExecutor<Eigen::	329	{ 163840 1 1}	{ 512 1 1}	12	16	2560	112.40	23.80	12547.69	11836.62	0.07

Figure 2.15 Vue présentant les compteurs de performance sous la forme d'un tableau. Chaque ligne montre différentes métriques à propos de l'exécution d'un noyau de calcul.

Finalement, de nombreuses informations essentielles sont disponibles, mais restent concentrées autour du processeur graphique. D'autres éléments, à propos du modèle flot de données pourraient être récupérées, à l'aide des marqueurs AMD et une instrumentation statique d'une ou plusieurs bibliothèques logicielles. Toutefois, les informations à propos du processeur central et du système d'exploitation manquent. Pour cela, une intégration des outils d'AMD dans une solution complète d'analyse de performance semble être une possibilité et paraît plus envisageable que dans le cas de Nvidia. Une limite demeure tout de même quant à la visualisation. CodeXL ne pourrait pas servir d'interface principale pour l'affichage des résultats, car les vues proposées sont fixes et ne peuvent ni être modifiées ou améliorées par un utilisateur. Ce manque de flexibilité nous pousse à considérer un autre outil pour l'analyse et la visualisation des traces. Enfin, l'utilisation du format ATP pour le stockage des traces semble également être un inconvénient à nos yeux, surtout du point de vue de la performance.

Outils libres

On retrouve majoritairement deux outils libres : `Tau` et `VampireTrace`.

Développé à l'université d'Oregon, `Tau` (Shende and Malony, 2006) est un outil pour l'analyse de systèmes utilisant les processeurs graphiques. Il permet une instrumentation du code source afin d'observer en détail le comportement des applications. Il intègre également un support pour présenter l'utilisation du processeur graphique faite par les applications ana-

lysées. Il est possible de récupérer l'ensemble des appels aux fonctions des bibliothèques utilisées pour programmer le GPU, comme OpenCL ou CUDA. Par ailleurs, les compteurs de performance peuvent également être collectés. Cela est permis par une utilisation interne des outils proposés par les constructeurs à cet effet, comme `CUPTI` pour Nvidia et `GPA` pour AMD.

Les résultats obtenus avec cet outil sont dans un format spécifique à Tau. Cependant, des solutions existent pour la conversion en format OTF (Open Trace Format) afin de pouvoir visualiser les résultats avec des outils comme `Vampir` ou `Paraver`.

`VampireTrace` (Müller et al., 2007), quant à lui, permet une analyse détaillée de l'exécution d'applications parallèles utilisant *MPI (Message Passing Interface)* ou *OpenMP*. Il s'agit d'un outil libre de sources développé à l'université de Dresden en Allemagne. Un support pour rendre compte de l'activité du processeur graphique est également disponible. Enfin, cet outil utilise le format OTF pour sauvegarder les traces générées. Il est donc possible d'utiliser des logiciels comme `Vampir` ou `Paraver` pour l'étape de visualisation des résultats.

Ces outils semblent intéressants puisqu'ils comprennent un support pour l'étude du processeur graphique. Néanmoins, les environnements d'analyse proposés semblent plus adaptés pour des applications hautement parallèles utilisant des technologies comme *OpenMP* ou *MPI*. Par ailleurs, il semble difficile d'intégrer à cela une analyse à un niveau d'abstraction plus élevé à propos du modèle flot de données.

Travaux de recherche

En dehors des outils existants et matures, des travaux de recherche ont également été menés dans l'optique d'analyser la performance du processeur graphique. Le premier travail, `CLUST` (Couturier and Dagenais, 2015), vise à fournir conjointement des traces présentant l'activité du processeur traditionnel et du processeur graphique. L'outil se base sur un mécanisme d'interception ainsi que sur certaines fonctions de profilage proposées par OpenCL afin de rendre compte des opérations réalisées par le GPU. Parmi ces fonctions on retrouve *clSetEventCallback()* et *clGetEventProfilingInfo()*. La première permet d'assigner une fonction de rappel à l'état d'une commande envoyée au GPU. Par exemple, il peut s'agir du début de l'exécution d'un noyau de calcul sur le processeur graphique. La seconde retourne des informations à propos d'un événement et peut être utilisée pour récupérer les temps de début et de fin de l'exécution d'un noyau de calcul.

Du point de vue de la collection des événements, `CLUST` utilise `LTTng` et bénéficie donc de ses avantages comme la génération de traces au format CTF. Cet outil offre un mode

d'utilisation très simple puisqu'il suffit de précharger une bibliothèque en utilisant la variable d'environnement LD_PRELOAD de Linux et d'activer une session de traçage `LTTng`. Cet outil semble donc très adapté pour notre projet de recherche et nous nous baserons dessus pour le cas où TensorFlow utilise le support SYCL.

Margheritta et Dagenais (Margheritta, 2017) ont proposé un autre travail `LTTng-HSA` pour l'analyse de performance de systèmes hétérogènes. Ce travail utilise également des techniques d'interception ainsi que des fonctions de profilage. Au contraire de `CLUST`, `LTTng-HSA` est destiné à la plateforme libre de sources ROCm développée par AMD et qui se base sur le standard HSA pour programmer le processeur graphique. Cet outil offre quatre cibles de traçage donnant des informations sur les appels de fonctions à la bibliothèque HSA, l'état des queues logicielles du GPU, les durées d'exécution des noyaux de calcul et des échantillons de compteurs de performance du processeur graphique. Par ailleurs, `LTTng` est utilisé pour la partie traçage et collection des événements. De ce fait, l'analyse et la visualisation des résultats seront flexibles et pourront être adaptées en fonction des besoins de l'utilisateur. Cet outil semble donc prometteur et certains de ses principes seront réutilisés au cours de notre projet de recherche.

2.4 Analyse et visualisation des résultats

Dans cette sous-section, nous allons nous intéresser aux moyens existants pour analyser et visualiser les traces collectées. Au niveau des outils ciblant les processeurs traditionnels, tels que `FTrace`, `Perf` ou encore `Strace`, des résumés statistiques sous forme de texte sont généralement proposés. Certains logiciels offrent des vues graphiques à partir de ces résultats textuels comme `KernelShark` ou `Flamegraph`. En ce qui concerne les outils ciblant les processeurs graphiques, les environnements de traçage et profilage proposés fournissent en général différentes analyses et vues. Malheureusement, il est en souvent impossible pour un utilisateur de modifier, d'adapter ou d'améliorer ces dernières en fonction de ses besoins. Tel que vu précédemment, notre choix s'oriente vers `LTTng` que ce soit pour l'analyse du processeur central ou graphique. Par conséquent, nous allons présenter les outils existants pour traiter et visualiser les traces au format CTF.

2.4.1 Babeltrace

Puisque `LTTng` génère des traces au format binaire CTF, les résultats ne peuvent pas être lus directement par une personne. L'utilisation d'un outil externe est donc nécessaire afin de pouvoir lire une trace. Le projet `Babeltrace`, qui contient l'implémentation officielle du

format CTF, répond à ce besoin. Il propose une bibliothèque permettant la lecture et l'écriture de traces CTF. Par ailleurs, cet outil offre également des possibilités de conversion dans les deux sens entre le format CTF et d'autres formats de traces.

`Babeltrace` est développé en C et propose donc une interface dans ce langage pour lire et écrire les traces. Dans l'objectif de simplifier son utilisation, une interface Python, qui offre des fonctionnalités similaires, a été développée. Cette solution s'intègre parfaitement dans un traitement a posteriori des traces. De nombreux cas existent pour lesquels il peut être nécessaire de lire une trace générée, de modifier certains éléments et de l'écrire à nouveau. Par exemple, on peut vouloir filtrer une trace en ne conservant que certains éléments, fusionner deux traces ensemble ou encore réordonner certains événements. La possibilité d'écriture d'une trace peut également être utile si l'on souhaite tester une technique de visualisation, en générant une trace artificielle contenant uniquement certains événements souhaités. `Babeltrace` peut donc être considéré comme un utilitaire aidant à traiter les traces après leur création. Bien que cet outil permette de lister sous forme de texte les événements contenus dans la trace, il ne peut pas être considéré comme un outil de visualisation. Dans cette optique, un autre outil majeur existe et est présenté dans la sous-section suivante.

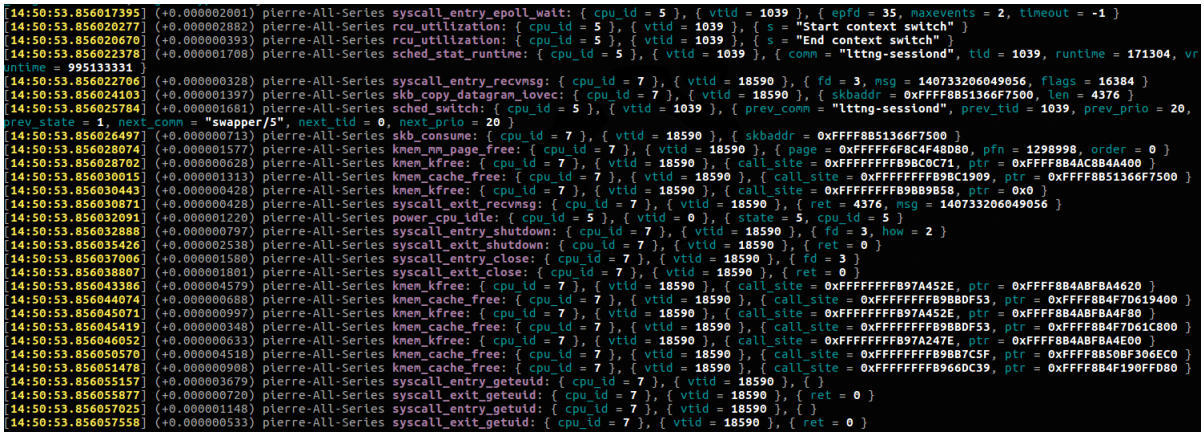


Figure 2.16 Exemple de trace du noyau Linux générée avec LTTng et visualisée avec Babeltrace.

2.4.2 Trace Compass

`Trace Compass` est un outil libre de sources pour l'analyse et la visualisation de traces de différents formats. Ce dernier est écrit en Java et se base sur l'environnement Eclipse. Par défaut, les vues proposées concernent la visualisation de traces du noyau du système d'exploitation avec la vue de *flux de contrôle (control flow view)* qui présente l'état de chaque processus s'exécutant sur le système. Une seconde vue importante appelée *vue des ressources*

(resources view) décrit l'état de chaque cœur du processeur central pendant toute la durée du traçage, ainsi que les interruptions rencontrées.

En tant qu'outil libre de sources, `Trace Compass` offre une très grande flexibilité. Un utilisateur peut aisément modifier et adapter des vues existantes, ou développer de nouvelles vues en fonction de ses besoins. La principale méthode pour cela consiste à développer une vue Java. Ce processus pouvant demander une certaine quantité de travail, une solution alternative appelée *analyse XML (XML analysis)* a été développée (Wininger et al., 2016; Kouame et al., 2015). Bien que moins complète, cette nouvelle méthode accélère et facilite le développement de nouvelles vues, ce qui est très appréciable dans un cadre de prototypage par exemple.

2.5 Profilage d'applications flot de données

Précédemment, nous avons abordé le traçage et le profilage des processeurs traditionnels et graphiques. Ces éléments sont essentiels dans le cadre de notre projet de recherche. Cependant, il nous reste à explorer les travaux antérieurs destinés à l'analyse de performance d'applications et de systèmes flots de données. C'est ce que nous souhaitons faire dans cette dernière section.

Un premier travail par Canale et al. (2014) se base sur les réseaux de Pétri et les graphes de traces d'exécution. Une transformation de ces graphes en un système linéaire dirigé par les événements est proposée. En particulier, le problème de la configuration optimale de la taille des différents tampons d'un modèle flot de données est abordé. Les cas de décodeurs JPEG et MPEG HEVC sont utilisés afin de démontrer l'efficacité de leur approche.

Janneck et al. (2008) proposent un travail permettant d'évaluer les performances d'un modèle flot de données. Les auteurs se basent sur la collecte de traces de causalités, puis d'analyses a posteriori pour présenter des résultats à propos de la consommation des entrées, pour chaque élément de calcul du modèle, ou encore sa latence totale.

Brunet et al. (2012a) introduisent quant à eux, une technique d'optimisation de programmes flot de données basée sur l'analyse a posteriori de traces d'exécution. Les auteurs se concentrent principalement sur le domaine du traitement de signal, en appliquant leur méthode à des décodeurs MPEG-4 SP et AVC/H.264.

Hoffmann et al. (2018) ont présenté un travail concernant l'analyse en ligne du chemin critique dans des applications flot de données distribuées. L'outil développé donne des informations et des résumés en temps réel à propos de la performance d'une application.

Mysore et al. (2008), proposent un autre travail qui consiste à collecter des informations à

propos de l'exécution par flot d'applications, pour ensuite analyser ainsi que visualiser les interactions entre les différents composants. Pour cela, les auteurs se basent sur un mécanisme de marquage des données, proche de techniques utilisées en médecine qui consistent à injecter certains composants radioactifs pour tracer des problèmes cardiovasculaires. Dans leur travail, un mécanisme d'étiquetage au niveau des instructions ISA est implémenté. Cela permet de collecter des informations à propos du noyau du système d'exploitation, des couches logicielles intermédiaires ainsi que des applications, sans aucune modification du code source. Cependant, ils expliquent que deux limites principales apparaissent. Tout d'abord, le surcoût introduit par leur méthode est assez important. De plus, les vues proposées dans leur travail ne sont pas suffisantes, ni pleinement adaptées. En effet, en raison du très grand nombre d'informations collectées, elles ne parviennent pas à mettre suffisamment en avant les problèmes potentiels de performance ou les limitations.

Enfin, un dernier travail vise à améliorer l'analyse de traces d'applications flots de données (Osmari et al., 2014). Les auteurs expliquent que la majorité des outils d'analyse fournissent des résumés statistiques standards, présentant trop d'informations concernant l'exécution d'une application. Ils ajoutent que trop de détails ayant un lien très faible par rapport à l'application analysée sont donnés et que cela n'apporte pas d'information utile pour un utilisateur. Dans leur travail, ils proposent une instrumentation du code source pour collecter des données, mais aussi un ensemble de visualisations. Ces dernières ont pour but de souligner réellement les caractéristiques importantes d'une application flot de données. Au travers de leurs analyses, les auteurs regroupent les nœuds du graphe flot de données en différents modules. En termes de vues, on retrouve notamment un diagramme de Gantt montrant l'utilisation des différents fils d'exécution, présentant la durée de traitement de chaque module. Une visualisation du graphe flot de données avec la circulation des données est aussi disponible. Par ailleurs, des diagrammes de dispersion sont proposés, et montrent l'évolution de la durée de traitement des différents modules ainsi que les délais liés à la communication au sein du graphe. Afin de montrer l'efficacité de leur travail, ils se basent sur HyperFlow, un système parallèle très léger ayant une approche flot de données. Ils démontrent que chacune des vues présentées est utile et permet effectivement de mettre en lumière un problème de performance.

Ces travaux sont très intéressants puisqu'ils soulignent le grand intérêt pour l'analyse de performance de modèles flot de données. Cependant, quelques manques apparaissent et laissent place à des améliorations et à d'autres projets de recherche. Beaucoup de travaux présentés se basent sur une analyse du modèle flot de données à un haut niveau d'abstraction, sans tenir compte des différentes bibliothèques logicielles intermédiaires, du système d'exploitation ou encore du matériel utilisé. Cela permet de détecter des problèmes de performance comme des

goulots d'étranglement ou de la congestion. Cependant, certains éléments manquent, comme l'assurance d'un taux utilisation élevé, voire optimal, du matériel disponible. Par ailleurs, lors de la détection d'un problème, les informations disponibles conservent un niveau d'abstraction élevé et ne sont pas toujours suffisamment détaillées pour comprendre réellement l'origine du problème. Bien souvent, les analyses existantes se basent sur une durée d'exécution trop longue d'un nœud du graphe pour détecter un problème au sein du modèle flot de donnée. Cependant, sans information bas niveau, proche des bibliothèques logicielles utilisées ou du système d'exploitation, il est difficile de solutionner le problème.

D'autres informations comme le suivi de la consommation mémoire, les transferts de données au niveau matériel ou encore les opérations d'entrée/sortie au niveau du système d'exploitation pourraient être utiles et permettraient d'améliorer l'analyse de performance du modèle flot de données. De plus, les travaux présentés se concentrent sur des cas d'applications utilisant seulement un processeur traditionnel. Au contraire, les modèles flots de données se destinent de plus en plus aux architectures hétérogènes combinant différents types d'unités physiques de calcul. Au cours de notre projet de recherche, nous viserons donc à apporter des solutions à ces limites.

2.6 Conclusion de la revue de littérature

Au travers de cette revue de littérature, nous avons pu évoquer différents sujets liés au travail présenté dans ce mémoire. Nous avons défini et présenté le modèle flot de données, en montrant également des cas d'utilisations de ce dernier. Nous avons justifié sa popularité avec l'émergence des architectures hétérogènes. Ces dernières ont également été abordées avec notamment le cas où un processeur graphique était utilisé afin d'aider un processeur traditionnel.

Après avoir présenté un historique des GPUs, leur architecture, ainsi que des exemples d'utilisation, nous avons expliqué les différentes solutions techniques pour leur programmation.

Nous nous sommes ensuite intéressés aux méthodes et outils existants pour le traçage et le profilage d'applications. Après avoir défini ces deux termes, nous avons eu l'occasion d'évoquer dans un premier temps les méthodes destinées aux processeurs traditionnels, avant de nous intéresser aux processeurs graphiques. Ces outils sont très importants pour notre travail, puisqu'ils représentent des solutions existantes afin de récupérer des informations à propos des deux types de processeurs. Nous avons vu que des outils à sources fermées étaient proposés par les constructeurs et offraient des fonctionnalités intéressantes. Cependant, leur manque de généralité, de flexibilité et la difficulté apparente à les intégrer avec d'autres outils, pouvaient

être un handicap. Au contraire, certaines solutions issues de travaux de recherches semblent prometteuses et très intéressantes dans notre contexte. Nous avons également évoqué certains outils permettant d'analyser et de visualiser les traces obtenues avec les techniques de traçage et profilage.

Enfin, dans une dernière section, nous avons évoqué des travaux existants dans le domaine de l'analyse de performance de systèmes ou d'applications flots de données. Ces derniers sont importants à considérer pour notre travail, puisqu'ils proposent des idées et solutions intéressantes avec toutefois quelques limitations.

CHAPITRE 3 MÉTHODOLOGIE

Le chapitre précédent a permis de présenter un état de l'art des différents domaines en lien avec le travail de recherche mené. Nous présentons maintenant la méthodologie utilisée pour notre travail. Nous détaillons les solutions logicielles choisies ainsi que les différentes configurations matérielles utilisées. Cela sera particulièrement intéressant pour comprendre les mesures de performance effectuées avec l'outil proposé, puisqu'il y a une dépendance forte avec le matériel. Par ailleurs, l'ensemble du travail présenté dans ce mémoire est libre de sources et disponible sur Github, ce qui facilitera la reproduction des résultats.

3.1 Utilisation de TensorFlow avec un processeur graphique

Au travers de ce mémoire, une technique de traçage et profilage d'applications flot de données utilisant un processeur graphique est proposée. Afin d'appliquer ce travail à un cas concret, la bibliothèque logicielle d'apprentissage machine et profond TensorFlow a été choisie. Utiliser cette dernière avec le support d'un processeur graphique dépend du type de matériel disponible. Comme la majorité des librairies d'apprentissage automatique et profond, TensorFlow ne supporte officiellement que les processeurs graphiques Nvidia. Cela se justifie par le fait que CUDA reste la solution la plus complète, performante et facile d'utilisation pour programmer un processeur graphique. Néanmoins, dans le cadre de ce travail, le caractère sources fermées des outils de Nvidia constitue un handicap. En effet, les techniques de profilage et traçage offrent plus de possibilités dans un environnement à sources libres, puisqu'une instrumentation du code source de l'application ou de la librairie est envisageable. C'est pourquoi nous avons décidé d'explorer de manière approfondie les possibilités d'utiliser TensorFlow avec des processeurs graphiques d'autres constructeurs. Finalement, trois solutions se sont dégagées :

— `TF-Coriander` : ce travail permet d'utiliser TensorFlow sur tous les processeurs graphiques supportant OpenCL 1.2. Il a été développé par Hugh Perkins et se base sur un compilateur (Perkins, 2017) qui va générer des noyaux de calcul OpenCL 1.2 à partir du code binaire des noyaux CUDA de TensorFlow. Ce travail bénéficie également de librairies logicielles spécialisées comme CLBlast (Nugteren, 2017), une version de BLAS compatible avec OpenCL et qui offre un maximum de performance pour des opérations d'algèbre linéaire effectuées sur des matrices et vecteurs. Malgré tout, `TF-Coriander` est basé sur une ancienne version de la librairie TensorFlow et les performances proposées restent limitées. D'un point de vue pratique également, il était moins intéressant

de développer un outil pour une version de TensorFlow non maintenue à jour et très peu utilisée.

— `TensorFlow avec SYCL` : il s'agit de la seconde solution existante pour utiliser TensorFlow avec du matériel provenant d'un autre constructeur que Nvidia. Cette option se base sur l'implémentation de SYCL proposée par Codeplay avec ComputeCpp. SYCL, au contraire d'OpenCL, offre un environnement de haut niveau en C++ et permet aussi de combiner, dans un même fichier, le code destiné au CPU et les noyaux de calcul pour le GPU. Ces deux points facilitent énormément le passage d'une base de code CUDA vers SYCL. Cette solution est en développement actif de la part des ingénieurs de Codeplay et Google, et à terme pourra être utilisée sur tout type de matériel.

— `TensorFlow avec la plateforme ROCm` : la troisième solution est basée sur la plateforme ROCm d'AMD. Il s'agit d'une suite logicielle libre développée dans le cadre du projet GPUOpen et visant le calcul à haute performance sur des architectures hétérogènes composées de différents types de matériel. Cette option utilise différents langages présentés dans la revue de littérature, comme HSA ou HIP, et offre le grand avantage d'être totalement libre de sources et documentée. Cela peut constituer un avantage certain, dans le cadre de notre recherche pour la compréhension du fonctionnement interne ainsi que les possibilités d'instrumentation du code source.

Ainsi, trois options différentes existent, en plus de la version officielle de TensorFlow, pour utiliser un processeur graphique. En se basant sur l'état actuel de chacune d'entre elles et de leurs avantages et inconvénients, nous avons décidé de nous concentrer sur la version de TensorFlow pour la plateforme ROCm. Cette solution semble prometteuse et a l'avantage d'être libre de sources, ce qui offre plus de possibilités et correspond bien à l'esprit du laboratoire de recherche dans lequel ce travail a été effectué. Cependant, trois raisons nous ont poussés à considérer également la version officielle de TensorFlow. Tout d'abord, nous avions accès à un processeur graphique Nvidia. De plus, l'idée était de fournir un outil pour l'analyse de performance le plus générique possible et le fait de pouvoir l'utiliser avec différents types de matériel est un avantage. Enfin, la version CUDA de TensorFlow reste la solution la plus utilisée, la plus avancée et proposant les meilleures performances. Finalement, nous avons également décidé de considérer la version de TensorFlow pour SYCL, puisque cette solution était toujours développée de manière active. Ainsi, l'outil proposé fonctionne avec trois versions différentes de TensorFlow : TensorFlow officiel, TensorFlow ROCm et TensorFlow SYCL.

3.2 Environnement de travail

En termes de configurations matérielles, nous avions accès à trois machines différentes. Deux d'entre elles sont très proches en termes de composants et utilisent le même processeur graphique AMD. Ces deux machines sont des ordinateurs de bureau tandis que celle avec un processeur graphique Nvidia est un ordinateur portable.

Tableau 3.1 Configuration matérielle

	Machine 1	**Machine 2**	**Machine 3**
OS	Ubuntu 16.04	Ubuntu 16.04	Ubuntu 16.04
Noyau Linux	4.13.0-32	4.13.0-32	4.4.0-130
CPU	Intel Core i7-4770 CPU 3.40GHz	Intel Core i7-4790 CPU 3.60GHz	Intel Core i7-6700HQ 2.60GHz
RAM	32 Go DDR3	32 Go DDR3	16 Go DDR4
GPU	AMD Nano R9 Fury	AMD Nano R9 Fury	GTX 950m
Mémoire vidéo	4 Go	4 Go	2 Go

En termes de logiciel, nous distinguons 5 cas. Deux versions différentes de TensorFlow pour la plateforme ROCm ont été utilisées. Le travail a été initialement développé pour la version 1.0.1 puis 1.3.0. Cela présentait une opportunité pour s'assurer que notre travail fonctionne encore pour une version plus récente de TensorFlow. La version 1.0.1 a également été installée sur deux machines différentes pour le cas d'utilisation de TensorFlow en mode distribué. La solution avec SYCL se base sur une version de TensorFlow relativement récente. Il s'agit de la même version que le TensorFlow officiel avec CUDA que nous avons utilisé. L'ensemble des installations de TensorFlow ont été effectuées par une compilation à partir des sources. Par ailleurs, les versions de LTTng et Babeltrace utilisées sont pratiquement les mêmes dans tous les cas.

Tableau 3.2 Configuration logicielle

	TensorFlow ROCm		**TensorFlow SYCL**	**TensorFlow CUDA**
TensorFlow version	1.0.1	1.3.0	1.6.0	1.6.0
Machine (tableau 3.1)	1	2	2	2	3
LTTng version	2.10.1 Kekriek	2.10.4 Kekriek	2.10.4 Kekriek	2.10.5 Kekriek
Babeltrace version	2.0.0-pre4			

3.3 Traçage et profilage avec la technique proposée

Dans cette partie, nous allons expliquer rapidement les différentes étapes nécessaires afin d'appliquer la méthode de traçage et profilage présentée.

Installation ROCm Afin de mettre en place la plateforme ROCm, il suffit de suivre les étapes d'installation proposées par AMD. Toutefois, notre technique repose sur une instrumentation de certaines de ses bibliothèques comme HSA, HC et HIP. Pour cela, il faut donc s'assurer d'utiliser une version instrumentée de ces dernières et de les compiler à partir des sources. Elles sont toutes disponibles sur Github.

Installation SYCL Peu de travail supplémentaire est nécessaire dans ce cas. Il suffit de s'enregistrer sur le site de Codeplay et de télécharger ComputeCpp. Il faudra néanmoins s'assurer d'avoir une version d'OpenCL contenant l'extension *cl_khr_spir*. Cette dernière est nécessaire pour utiliser ComputeCpp et est disponible avec le pilote propriétaire d'AMD : AMDGPU-PRO pour OpenCL.

Installation de TensorFlow Puisqu'une instrumentation du code source est utilisée pour de collecter des informations importantes quant à l'exécution d'une application, il est nécessaire d'avoir une version particulière de cette librairie. Pour cela, nous avons effectué une copie, ou "fork" en anglais, de chaque dépôt de TensorFlow : TensorFlow ROCm 1.0.1, TensorFlow ROCm 1.3.0, TensorFlow SYCL et TensorFlow CUDA. Ces dernières sont disponibles sur Github, et il suffit donc de récupérer leur code source en s'assurant utiliser la branche

contenant notre instrumentation. L'installation à partir des sources nécessite l'utilisation de Bazel et il faudra s'assurer que sa version soit compatible avec la librairie TensorFlow que l'on souhaite compiler. Les différentes étapes pour compiler TensorFlow sont décrites dans la documentation officielle de la librairie.

Traçage Une fois les étapes précédentes faites, TensorFlow devrait fonctionner et parvenir à utiliser le processeur graphique. Avant de lancer l'application que l'on souhaite analyser, il est nécessaire de réaliser deux étapes. Tout d'abord, différentes variables d'environnements doivent être assignées. Cela est nécessaire uniquement pour TensorFlow avec la plateforme ROCm et SYCL et dépendra des informations que l'on souhaite collecter ainsi que de la technique utilisée. Voici un descriptif des principaux cas :

- Traçage de l'API HIP : `HIP_PROFILE_API` doit être assigné à 2.
- Récupération des temps des opérations réalisées par le GPU :
  - Par l'instrumentation de HC ou par l'analyse des logs : `HCC_PROFILE` doit être assigné à 2.
  - Par interception : `HSA_TOOLS_LIB` doit contenir `libhsa-runtime-tools64.so.1` et `LD_PRELOAD` doit contenir le chemin jusqu'à la librairie d'interception `hsa_kernel_times.so`
- Collection des compteurs de performance : `HSA_TOOLS_LIB` doit à nouveau contenir `libhsa-runtime-tools64.so.1`, `HSA_EMULATE_AQL` doit valoir 1 et `LD_PRELOAD` doit contenir le chemin jusqu'à la librairie `hsa_perf_counters.so`
- Traçage de l'API OpenCL et récupération des temps des opérations asynchrones faites par le GPU : `LD_PRELOAD` doit contenir le chemin jusqu'à la librairie d'interception `libCLUST.so.0.0.0`

Une fois l'assignation des variables d'environnement faite, il reste à configurer LTTng. Pour cela, il suffit activer tous les points de traces correspondant aux informations que l'on souhaite collecter. Ces deux étapes peuvent être réalisées manuellement par l'utilisateur, mais il reste plus simple d'utiliser un des scripts développés.

Arrêt du traçage Il s'agit simplement d'arrêter le traçage LTTng.

Traitement a posteriori Après avoir collecté l'ensemble des événements, il est nécessaire d'effectuer un traitement de la trace. Cette étape est décrite en détail dans l'article à la section 4.4.3 et se fait simplement à l'aide d'un ou plusieurs scripts.

3.4 Code source du travail

L'ensemble du travail développé au cours de projet de recherche est disponible sur Github. Les quatre versions de TensorFlow sont des copies des dépôts officiels de la librairie et l'ensemble des scripts et librairies nécessaires au traçage, au profilage et à l'analyse sont disponibles dans un dépôt commun appelé `TensorFlow-profiler`. De même, les versions instrumentées des bibliothèques de la plateforme ROCm ont également été copiées et ajoutées sur Github. Enfin, Babeltrace a pu causer quelques problèmes, c'est pourquoi une version modifiée, mais fonctionnelle, est également disponible sur Github.

— TensorFlow profiler : `https://github.com/pzins/tensorflow-profiler`
— TensorFlow ROCm version 1.0.1 : `https://github.com/pzins/hiptensorflow`
— TensorFlow ROCm version 1.3.0 : `https://github.com/pzins/tensorflow-rocm`
— TensorFlow officiel 1.6.0 : `https://github.com/pzins/tensorflow`
— TensorFlow SYCL 1.6.0 : `https://github.com/pzins/tensorflow-sycl`
— HIP : `https://github.com/pzins/HIP`
— HC : `https://github.com/pzins/hcc`
— HSA (ROCR-Runtime) : `https://github.com/pzins/ROCR-Runtime`
— Babeltrace : `https://github.com/pzins/babeltrace`

Pour l'ensemble de ces dépôts, il est important d'utiliser la branche de travail appelée **lttng**.

Dans ce chapitre, nous avons présenté la méthodologie employée pour notre travail. Nous avons abordé tout d'abord certains points importants afin d'utiliser TensorFlow avec des processeurs graphiques. Par la suite, nous avons précisé notre environnement de travail d'un point de vue matériel mais aussi logiciel. Enfin, une dernière sous-section énumérait les différentes étapes nécessaires pour appliquer notre méthode. Le chapitre suivant constitue un article de recherche dans lequel nous exposons en détail notre solution. Nous présentons son fonctionnement et démontrons son efficacité à partir de plusieurs cas d'utilisation.

CHAPITRE 4 ARTICLE 1 : TRACING AND PROFILING MACHINE LEARNING DATAFLOW APPLICATIONS ON GPU

Authors
Pierre Zins <pierre.zins@polymtl.ca>
Michel Dagenais <michel.dagenais@polymtl.ca>
Department of Computer and Software Engineering ; École Polytechnique Montréal

Submitted to International Journal of Parallel Computing

Keywords dataflow, GPU, machine learning, performance analysis, profiling, TensorFlow, tracing

Abstract
In this paper, we propose a profiling and tracing method for dataflow applications with GPU acceleration. Dataflow models can be represented by graphs and are widely used in many domains like signal processing or machine learning. Within the graph, the data flows along the edges, and the nodes correspond to the computing units that process the data. To accelerate the execution, some co-processing units, like GPUs, are often used for computing intensive nodes. The work in this paper aims at providing useful information about the execution of the dataflow graph on the available hardware, in order to understand and possibly improve the performance. The collected traces include low level information about the CPU, from the Linux Kernel, as well as high-level information about the dataflow model. This is followed by post-mortem analysis and visualization steps in order to enhance the trace and show useful information to the user. To demonstrate the effectiveness of the method, it was evaluated for TensorFlow, a well-known machine learning library that uses a dataflow computational graph to represent the algorithms. We present a few examples of machine learning applications that can be optimized with the help of the information provided by our proposed method.

4.1 Introduction

Achieving very high performance is an essential aspect of computing systems. The improvement in traditional CPU performance has recently slowed down. Indeed, techniques used to improve the performance like increasing the CPU clock frequency and adding more memory cache have reached a limit. Therefore, newer improvements largely consist in using highly

parallel computing environments to achieve better performance. Heterogeneous architectures have emerged, in which traditional CPUs get support from co-processing units like GPUs, FPGAs or DSPs. With the appearance of General Purpose GPU (GPGPU), GPUs are no longer only intended for graphical applications (Owens et al., 2008). They are increasingly used for general purpose computations.

In order to benefit from these new parallel architectures, the dataflow model has become very popular. It is a data-centric model and can be represented as a graph, where nodes are the operations that are applied to the incoming data, and edges represent the flow of data. It is used in different domains like signal processing (Boutellier et al., 2018; Boutellier and Nyländen, 2017; Bezati et al., 2010; Hentati et al., 2012) and image (Blattner et al., 2017) or video processing (Bourrasset et al., 2016b). New languages based on this approach have also been developed (Halbwachs et al., 1991; Caspi et al., 1987; Wadge and Ashcroft, 1985) and (Eker and Janneck, 2003). They are inherently parallel and therefore well adapted to heterogeneous platforms. In the machine learning domain, Krizhevsky et al. (2012) demonstrated the advantages of using GPUs to implement deep learning models for images. Consequently, the dataflow model became popular in this field as well and libraries like Theano (Bergstra et al., 2010; Al-Rfou et al., 2016) and TensorFlow (Abadi et al., 2017, 2016a,b) use it to jointly program the CPU and GPU. In order to insure maximum performance, we need to be able to analyze the execution of the dataflow application. The challenge here is to evaluate the execution on all the available hardware and insure that we maximize their usage.

Tracing and profiling are two techniques that can help to identify several types of problems, like deadlocks, bugs, bottlenecks or inefficient hardware usage. Applying them to the CPUs cores, either on the kernel side or the user side, is a task already well explored and has demonstrated its efficiency in the past (Goulet, 2012; Fournier et al., 2009). However, in the case of heterogeneous architectures, it is not sufficient, as we also need to collect information about the co-processing units used.

In this work, we focus on the case where the CPU gets support from a GPU. Profiling a GPU is usually possible thanks to GPU vendors tools like CodeXL for AMD, Nsight for Nvidia and Graphic Performance Analyzer for Intel. They target specifically the GPU and do not provide information about the CPU. For that, the integration with an external element or method is necessary but seems difficult, because the internal characteristics of these tools are not documented or published. The offered visualizations are also specifically defined and cannot be modified. Moreover, it is not possible to develop new views according to the user

needs. Furthermore, each tool is limited to the hardware of a specific vendor. Therefore, it is not possible to achieve a general solution based on one of these tools that can work with GPUs from different vendors. Finally, the goal of profiling and tracing dataflow applications is to propose a unified view, gathering information from every device, as well as developing specialized views focused on some key points like the memory consumption or the memory transfers between the devices. The flexibility required for that purpose is not available in the tools proposed by the GPU vendors.

In this work, we evaluated our method with TensorFlow. This library uses an efficient dataflow approach and is designed for heterogeneous architectures, and GPUs in particular. The choice was also motivated by its very high performance and popularity in the machine learning domain. In this paper, we propose a profiling and tracing method for dataflow applications that use a GPU. Several elements are analyzed and we obtain, as a result, a detailed trace of the execution of the dataflow application. We maintain a very low overhead, to minimize the impact on the studied application. We developed appropriate views and analyses that are highly beneficial for users.

The originality of our work resides in the combination of information from several layers and from all the computing units involved. In our approach, we combine low-level information related to the Operating System or the GPU, with mid-level information from several libraries, and high-level knowledge about the dataflow model. As a result, we obtain a trace containing a rich set of events. Different analyses and visualizations allow users to obtain useful information in order to guide their optimization efforts.

Our main contributions are :

- A multilevel data collection method for dataflow applications that execute jointly on CPU and GPU.
- Trace analyses and visualizations to show the results of the tracing and profiling process and to detect performance issues and limitations.
- An implementation of our method for the machine learning library TensorFlow.
- Several use cases to demonstrate the efficiency of our work to understand and improve the performance of TensorFlow applications.

The article is organized as follows. First, we describe existing work in the domain of CPU, GPU and dataflow systems tracing and profiling. Then, we explain the different parts of the method. In section 3, we apply our method to different TensorFlow applications. We show in particular how the views obtained can help a user to understand the performance and suggest optimizations. After that, we evaluate the proposed method for TensorFlow by computing

its overhead on different platforms. Finally, we conclude and suggest some possible directions for future work.

4.2 Related work

The related work section is composed of three subsections. We start by describing existing work on tracing and profiling the CPU. Thereafter, we cover GPU profiling and, finally, we look at existing work about dataflow model analysis.

4.2.1 CPU tracing and profiling

Tracing is a technique that consists in recording some events during the execution of an application. This process aims to impose a minimal overhead, compared to a normal execution of the application. Obviously, this depends strongly on the number of collected events. Tracing is well-known for performance analysis and several tools have already been developed in this regard. Most of them are intended to work on GNU/Linux. Indeed, tracing often involves a static instrumentation of the source code and, therefore, open-source projects are more suitable. In spite of being more limited, tracing in closed-source environments is also possible, like ETW for Windows. Profiling is another technique for performance analysis and corresponds to the process of gathering some metrics about the execution of an application. For example, monitoring the memory usage of a device, or sampling hardware performance counters refer to profiling. In the next paragraphs, we present the main tools for CPU tracing and profiling.

Strace

`Strace` is a Linux tool that can trace all the system calls as well as the signals received by a process. This tool is relatively popular due to its ease of use. Therefore, it is a powerful option for troubleshooting Linux systems and is often chosen by system administrators. However, its poor performance and the high overhead introduced is a major drawback.

Perf and Ftrace

`Perf` and `Ftrace` are two solutions integrated directly into the Linux Kernel for performance analysis.

With `Perf`, the entire system (userspace and kernel space) can be statistically profiled. For example, performance counters from the CPU can be collected. Static instrumentation of the

source code and dynamic instrumentation using *kprobes* and *uprobes*, respectively for kernel and userspace tracing, are also available.

`Ftrace` (Rostedt, 2009) is a tracing framework for the Linux Kernel and contains several tracers to collect different information. Originally designed for static instrumentation, new features were gradually added, including dynamic tracing of kernel functions using *kprobes*. Filtering is also available and allows the user to collect only some specific information. Ftrace is managed with a special file system, named TraceFS.

LTTng

`LTTng` is a Linux tool for kernel side and userspace side tracing (Desnoyers and R Dagenais, 2006). Unlike `Ftrace` and *Perf*, its kernel tracer module is loaded at the start of the tracing subsystem and is not part of the Linux kernel. `LTTng` has been developed to support the tracing of multithreaded applications. The overhead introduced is lower than with other tools. Collecting events with LTTng, produces a trace in CTF format (Common Tracer Format). This standardized binary format was developed with the objective of optimizing the trace writing performance and the compactness. `Babeltrace` is the main tool to read and write CTF traces, and Trace Compass offers a flexible visualization and analysis environment. Due to its ease of use and its high-performance multi-level tracing capabilities, we used `LTTng` in this work.

4.2.2 GPU tracing and profiling

Tracing GPUs consist in two main parts. The first part provides information about the GPU activity but is performed on the CPU. It consists in collecting all the function calls to the GPU libraries like CUDA or OpenCL. This part can be performed by some of the tools described previously. The second part consists in gathering the timestamps of every GPU related event : GPU kernels, memory copies, and synchronization barriers. Profiling the GPU is also possible by collecting performance counters. The latter represent useful metrics that can give insights into the performance of a GPU kernel function executed on a GPU.

Nvidia nvprof and Nsight

`Nsight` is a performance analysis environment developed by Nvidia for their GPUs. It offers debugging but also profiling capabilities with `nvprof`. The latter can be used inside `Nsight` or directly in a command-line version. A lot of information about the GPU can be collected : CUDA API function calls, the beginning and end times of asynchronous events like GPU

kernels or memory copies and performance counters. Nvidia is also going to release shortly three new tools for performance analysis. The first one is a system-wide performance analysis tool called `Nsight Systems`. `Nsight Compute` is the second one and focuses on GPU kernels profiling. The last tool is `Nsight Graphics` and concerns graphics applications.

Nvidia tools present several limitations. First, the results of a profiling session are in a binary and proprietary format that is intended to be used within the `Nsight` environment only. Moreover, these tools focus on the analysis of the GPUs, which is insufficient in our context. Finally, the visualizations offered are strictly defined and cannot be modified, extended or improved by users.

CodeXL

`CodeXL` is a suite of profiling tools designed for AMD hardware. Like `Nsight`, this tool offers profiling and debugging opportunities. It is available as a standalone application on Linux and also as a Visual Studio extension on Windows. `CodeXL` offers several features like OpenCL API profiling, APU/CPU/GPU power profiling, performance counters collection and graphics API tracing (OpenGL, Vulkan, and DirectX). Internally, it uses different tools and libraries like `RCP (Radeon Compute Profiler)` or `GPA (GPU Performance API)`.

Like the Nvidia tools, they target exclusively the GPUs, and the proposed views cannot be modified. Therefore, CodeXL lacks flexibility and it might be difficult for a user to integrate it into a tracing and profiling environment.

4.2.3 Dataflow profiling

In the two previous subsections, we described existing tools for tracing and profiling the CPU as well as the GPU. This subsection presents work focused on analyzing dataflow models. Canale et al. (2016) described a new method based on Petri nets for optimizing and representing dataflow programs. A heuristic algorithm has also been developed to find an optimal buffer size for data streams. This approach provides a relatively high level analysis of dataflow programs, and they demonstrated its efficiency to optimize well-known streaming applications like JPEG or MPEG decoding. Jörn et al. (2008) presented some techniques for analyzing the execution of dataflow programs. They introduced the notion of causation traces, as well as some analysis techniques that can be applied. As for the previous work, they illustrated the benefit of their technique with streaming applications like MPEG-4 decoders. Brunet et al. (2012b) explained another dataflow trace analysis to optimize signal processing algorithms. The effectiveness of the technique is demonstrated with two examples : MPEG-4

SP and AVC/H.264 video decoders. Mysore et al. (2008) developed a data tagging method at ISA level. The latter can be compared to medical techniques that consist in injecting a radioactive substance into a human to detect heart disease, for example. Their work is focused on dataflow systems and allows a user to collect information at different layers, from the kernel of the operating system up to the applications. However, the introduced overhead is problematic, and the visualization of the results is a challenge because of the amount of information collected. Finally, Osmari et al. (2014) described Smart Trace, a trace analysis tool specialized for parallel and especially dataflow traces. The authors presented a set of dataflow-centered visualizations and analyses and showed the efficiency of each.

4.3 Background

In this section, we start with a presentation of TensorFlow. We quickly explain some principles and key elements of this library. We also mention some technical constraints and limitations when using it with a GPU.

4.3.1 TensorFlow concepts

This research work aims at profiling machine learning dataflow applications that use GPU acceleration. The proposed method is general but, in order to apply the work on a concrete example, we decided to use TensorFlow. It is therefore useful to explain some concepts about this library.

Developing an application with TensorFlow usually involves two distinct steps. First, the user combines several operations to create the computation graph of the model. The second step is just the execution of this graph. When training a model, a backward pass is usually involved in order to update the weights of the model. From a dataflow perspective, this pass is not problematic and simply represents additional nodes in the graph. Globally, the training process requires many executions of the model with different data as input.

TensorFlow uses a specific object called *Session* to manage the graph. This object offers a method named *Run* that can trigger one execution of the graph. From a practical point of view, the user calls many times the *run* method of the unique *Session* object, each time with new input data. Each call starts one execution of the graph with the provided input data. Thus, the whole training process is composed of a succession of graph executions. Our work mostly targets the training step, as it is the most demanding in terms of computation and duration, but the inference step can also be analyzed just as well. The only difference with the training step is the absence of the backward pass, which has no effect it terms of tracing

or profiling.

4.3.2 TensorFlow with a GPU

Using TensorFlow with GPUs adds some technical constraints. Indeed, as the vast majority of machine learning libraries, the official support for GPUs in TensorFlow is done with *CUDA*, which means that it is only usable with Nvidia hardware. Two main efforts exist in order to support GPUs from other vendors. They are still in development and not entirely up-to-date with the official version of TensorFlow.

The first possibility concerns AMD GPUs, through ROCm, an open-source platform for GPU-enabled high-performance computing developed by AMD (Stoner, 2016). This platform is based on *HSA (Heterogeneous System Architecture)* (Rogers, 2015), a hardware and software stack that allows the different computing units, like CPUs or GPUs, to cooperate efficiently. Several other open-source libraries from AMD are used like *HIP (Heterogeneous-Compute Interface for Portability)*, a CUDA-like single source C++ API for GPU programming, and *HC (Heterogeneous Compute)*, a lower-level C++ API for accelerated GPU computing.

The second option is more general and consists in a new SYCL backend for TensorFlow (Goli et al., 2017). SYCL (Keryell et al., 2015) is a specification from the Khronos Group that provides an abstraction layer on top of OpenCL. It brings several improvements like the support for modern C++ and the single source programming feature. This solution is supposed to work with a wide variety of GPUs and is not limited to AMD or Nvidia GPUs.

Apart from some technical details, the general idea remains the same, on all platforms and with the three implementation versions of TensorFlow. In the next section, we present the different parts of the proposed method.

4.4 Proposed method

In this section, we discuss the concepts and principles of the proposed method. We start by describing our architecture. After this, we present all the information we decided to collect about the execution of the dataflow application. Once all the data has been collected, we address the trace post-processing steps. Finally, the analyses and views to visualize the results are presented.

4.4.1 Architecture

Figure 4.1 shows the architecture we used in our work. The first element on the left represents the multilevel data collection. Information about different components is collected using instrumentation, tracing and profiling techniques. We can distinguish three main levels :

— Application level : the dataflow model (TensorFlow)
— Middle level : the libraries and APIs used to program the GPU (CUDA, OpenCL, HIP, HSA)
— Low level : the computation units (CPU with the Linux Kernel and GPU)

After the tracing and profiling process, the collected events are post-processed to create a fully coherent trace. Finally, the results can be statistically analyzed or visualized. Both options help the user to understand the execution of an application and to improve its performance. In the next subsections, we detail each part of the architecture.

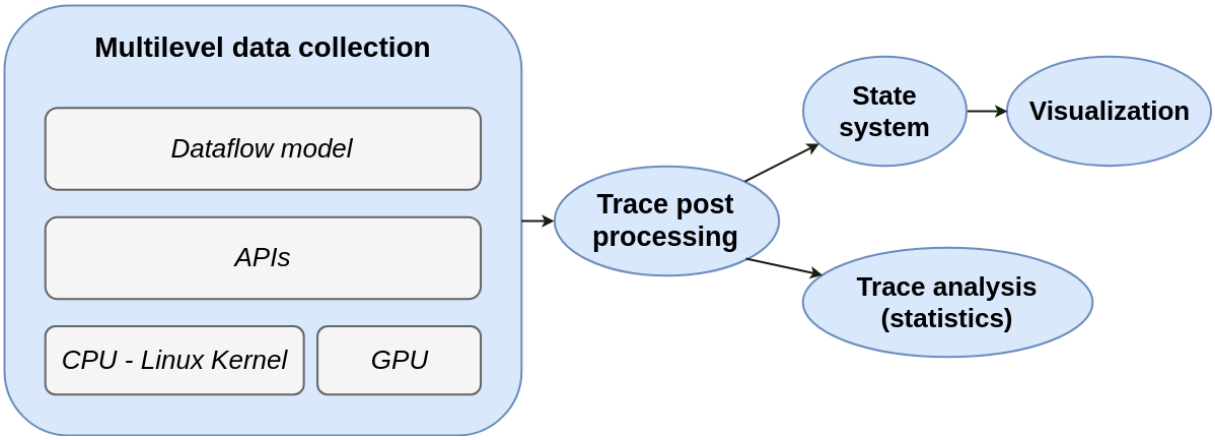


Figure 4.1 Performance analysis architecture. Information about different components is collected at several levels. The resulting trace is post-processed and then the results can be statistically analyzed or graphically visualized.

4.4.2 Multilevel Data Collection

Our approach is based on the collection of many information about the execution of an application on an heterogeneous architecture. For that, we propose a multilevel data collection method composed of different elements.

TensorFlow instrumentation

Dataflow applications can usually be seen as a graph composed of nodes that represent operations, and edges on which the data flows. In order to profile efficiently those types

of applications, we need to have access to the graph. Therefore, an instrumentation of the library that implements the dataflow model is necessary. In this paper we evaluate our method with TensorFlow. Therefore we describe different key parts of it but the general idea of the instrumentation can easily be ported to other libraries.

From a technical point of view, we opted for LTTng for its efficiency and flexibility. Moreover, the resulting trace is in the binary Common Trace Format (CTF) which aims at high performance, especially for writing the trace. When tracing an application, events are recorded each time an LTTng tracepoint is hit. Therefore, the instrumentation is essential and the tracepoints should be inserted carefully at strategic locations in the source code of TensorFlow. We discuss the main elements that should be profiled in the case of a dataflow application.

Session In a dataflow application, the execution consists in feeding some data to the input nodes of the graph which triggers the execution of every node in the graph until the output nodes. Every time a node has received all its inputs, it can start computing. Feeding data to the graph, and then retrieving the final results, can be considered as one iteration. Since in a dataflow application new data is continuously fed to the graph, it is necessary to have a general insight of the current state of the dataflow model. Knowing the beginning and the end of every iteration is therefore essential. In TensorFlow, an iteration corresponds to one call to the *Run* function of the *Session* object, as described in subsection 4.3.1. Therefore, our instrumentation deals with these two elements.

Operations In order to profile dataflow applications, we need to trace the different operations that process the data flow. Indeed, they define the behavior of the application. One operation, represented by a node in the graph, performs the same processing continuously on the incoming data. Four main informations should be collected about operations :

1. the processing beginning time
2. the processing end time
3. the device to which the operation is assigned. In this work, two options are possible, either the CPU or the GPU.
4. the type of the operation : synchronous or asynchronous.

Usually, most of the computation nodes are assigned to the GPU, because of its higher computation capabilities. In this case, little work is executed on the CPU : preparing the execution with data transfers for example and sending work to the GPU. All the computation happens on the GPU and therefore this component should constitute a priority in the profiling

process. The internal instrumentation of the GPU access library is not sufficient to collect the real execution times of the operations executed on the GPU, and this issue is discussed in subsection 4.4.2.

Scheduling In order to understand the behavior of a dataflow execution, it can be interesting to have an insight into how all the operations in the graph are scheduled. Indeed, in many cases, including TensorFlow, several nodes are assigned to the same computing unit and can be ready at the same time. It is therefore important to know how the library decides which node to execute.

Collecting all the beginning and end timestamps of every operation in the graph already brings some information about the scheduling, but we can go deeper. To do that, we need to understand the inner working of TensorFlow, and especially how it schedules the nodes. This part is totally dependent on the library used to implement dataflow applications.

The TensorFlow scheduling mechanism uses a threadpool and two different types of queues : *ready* and *inline_ready*. When a node finished its execution, the result is available and the successors of the node in the graph are activated. When a new node is ready, it is first enqueued into the *ready* queue and TensorFlow continuously schedules the nodes from it. Two options are possible. If the node selected from this queue is expensive, it is dispatched to the thread-pool in a new thread to be executed. Otherwise, the node is considered as inexpensive and is put into the *inline_ready* queue to be executed directly within the current thread. TensorFlow usually considers nodes executed on the GPU as inexpensive, because they consume very few resources on the CPU where the scheduler runs.

In terms of instrumentation, it represents three elements. First, we want to know the nodes scheduled from the *ready* queue. Secondly, we would like to follow all the operations executed from the *inline_ready* queue. Finally, it can be interesting to know which nodes are activated by each completed node.

Memory The data-centric characteristic of a dataflow model makes memory usage another essential aspect. In this model, data is continuously fed to the graph and flows along the graph edges. As a result, we need to have information about memory usage. This is even more crucial in the case of machine learning applications in which the data considered is usually tensors (multidimensional matrices) that can be very large. It is important to have information about this for the CPU cores, but even more for the GPUs as their memory is a relatively scarce resource, typically much smaller than the CPU RAM. Two elements are analyzed :

— Memory allocations and deallocations

— Memory transfers between the devices

In order to use a GPU, a developer generally needs platforms, libraries or frameworks like CUDA or OpenCL. All of them provide some API functions in order to program the GPU. In particular, a few functions offer the possibility to allocate memory on the host (CPU) or on the device (GPU) and to transfer data between them. The idea here is to locate all the calls to these specific functions in the analyzed application and to instrument them. If the application provides an internal wrapper around these functions like in TensorFlow, the instrumentation process becomes easier. If we consider memory allocations and deallocations, we usually deal with four different functions :

1. Allocate on the Host
2. Deallocate on the Host
3. Allocate on the Device
4. Deallocate on the Host

Instead of calling these functions very frequently, many applications choose to allocate all the available memory on the device at initialization time. Then, the allocated memory is simply managed by an internal allocator. This depends on the library that implements the dataflow model.

If we consider TensorFlow, it uses an internal allocator called *BFC Allocator* that implements the *Best Fit with Coalescing* algorithm. This allocator is used for every device (CPU or GPU), and simply adds another level of instrumentation, as we need to find all the functions related to memory allocation and memory deallocation within this allocator. Inside TensorFlow, the *BFC Allocator* is a simple version of the *Doug Lea's malloc (dlmalloc)* (Lea, 1996) and splits all the memory into several chunks. They represent a memory fragment of a specific size. Moreover, chunks are stored into bins which are collections of similar-size free chunks. This allocator is supposed to keep fragmentation to a minimum. Thus, in order to have a finer-grained analysis of memory management inside TensorFlow, we instrumented this allocator. As a result, we can bring three statistics out.

1. Global statistics for the allocator, with several metrics : the number of bytes in use, the number of allocations, the maximum number of bytes in use and the largest allocation.
2. Chunks statistics which monitor the chunks usage. Several metrics are available : the total number of chunks, the number of used chunks and the number of free chunks. It also follows the number of bytes used and, unlike the global statistics, it can differentiate the number of bytes used and the number of bytes requested. The difference can be explained by the restrictions for the chunk sizes to certain values, which can be

bigger than the real allocation request. With that information, we can also compute the number of wasted bytes.

3. Bins statistics : These monitor all the bins by giving the number of chunks in each bin and the corresponding number of bytes.

As mentioned before, memory transfers between devices are also important. When using GPUs, a memory transfer is considered as a command sent to the device to either read or write data. The first case is for memory copy from the Device (GPU) to the Host (CPU) and the second represents a memory copy from the Host (CPU) to the Device (GPU). Moreover, they are usually performed asynchronously by the DMA unit of the GPU, in order to overlap with the computation and consequently hide some latencies. To track them, we collect all the calls to the API functions that cause memory transfers. As for memory allocation, we need to locate all the calls to the corresponding API functions and instrument them. The instrumentation here has two purposes :

1. Getting the duration of the call by adding a tracepoint just before and just after the call
2. Getting the amount of data involved in the memory transfer

If the memory transfer is performed synchronously, the duration obtained actually corresponds to the time spent to copy the data between the two devices. However, if the copy is asynchronous, the duration only represents the time required to launch a command to the GPU and not the actual duration of the transfer. Getting the real duration of the memory transfers is addressed in section 4.4.2.

GRPC In this work, we focused on dataflow models executed on a single machine. However, it is also possible to distribute the execution on several machines. In this case, the graph is partitioned on several hosts. In addition, each host is free to offer a GPU to accelerate the execution of certain dataflow nodes. Profiling distributed dataflow models is possible with local profiling on each machine, and the possibility to jointly visualize the traces from several machines. In order to enhance the information and to highlight some key points, we also instrumented the code that allows the dataflow models to be distributed on several machines. In the case of TensorFlow, the implementation is based on two elements : gRPC, an open-source and high performance RPC framework, and protocol buffers, a serializing tool to encode the data exchanged over the network. During the execution, if two connected nodes are assigned to two different machines, a tensor request is sent by one machine and the second machine computes the tensor and sends the response back to the first machine.

Figure 4.2 presents an example of a graph with 7 nodes shared among two machines. The

first machine can process all the nodes until node E, as they do not require any external element. However, to compute node E, the first machine needs the output of node G which is assigned to the second machine. Therefore, to continue the execution, machine 1 sends a tensor request to machine 2. Once the request has been received, machine 2 starts the computation to get the result from node G and then sends it back to machine 1. Nodes F and G could be computed in advance by machine 2, or only when their results are required. This is mainly the difference between *as soon as possible* and *as late as possible* scheduling. In the first case, as soon as all the inputs of a node are ready, it is computed, whereas in the second case, the node is executed only when its output is required.

Finally, in this work, we proposed an instrumentation that shows the time spent by one machine waiting for a tensor from another machine. Moreover, we are able to track the size of the request and the tensor response exchanged by the machines over the network.

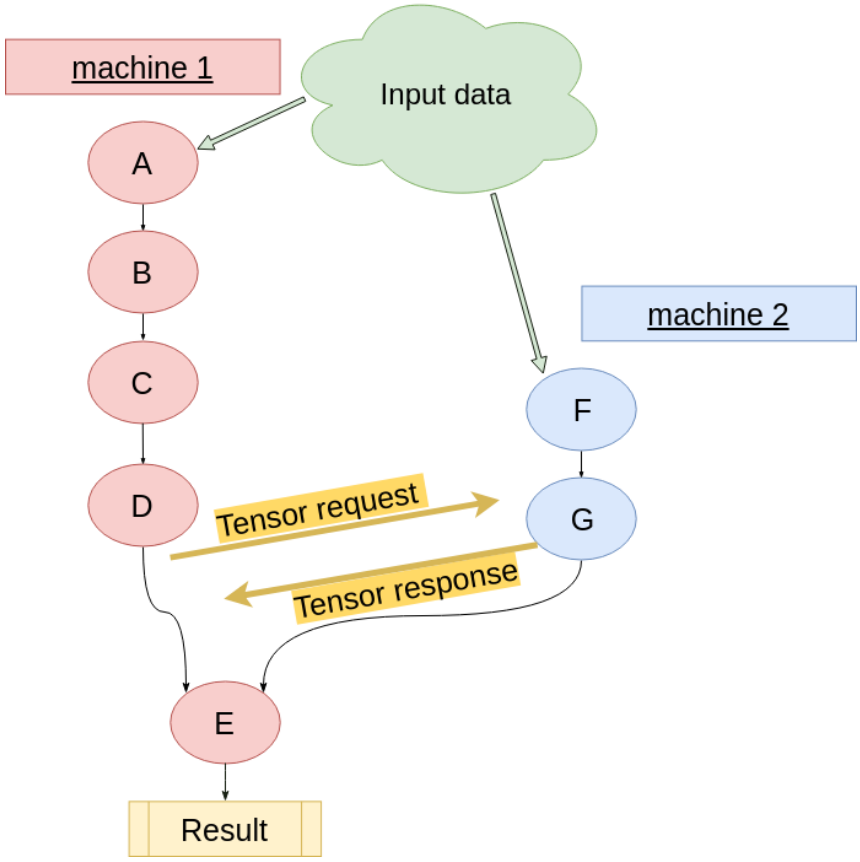


Figure 4.2 Distributed dataflow example : the graph is shared among two machines and data is fed to two input nodes (A and F). The computation of node E (assigned to machine 1) needs the result of node G (assigned to machine 2). Therefore, a tensor exchange between the machines is required.

API tracing

As we have seen in the previous subsection, a static instrumentation of the application that provides the dataflow model is necessary to get information related to the dataflow model itself. However, it is not sufficient as information about the GPU activity is missing. In order to have a general method to be aware of the GPU usage, we decided to trace the API of the library used to program the GPU. This brings important information about the GPU usage by the dataflow model. When a node performs work related to the GPU, we want to know which API functions were called and how long they took. Three methods exist for that :

1. Profiling callbacks : this consists in setting an asynchronous callback called at the end of each API function. This allows to get the name of the function as well as the start and end time of the call. For example, CUPTI, a profiling library from Nvidia, provides this type of callback for CUDA, and allows a user to programmatically collect API function timestamps and durations.
2. API instrumentation : this method requires access to the library used to program the GPU. In this case, adding two tracepoints to each function of the API is sufficient to collect the beginning and end time of the API call. For example, this can be implemented for the HSA API into the runtime of the ROCm platform.
3. The last possibility is based on interception and is quite similar to the previous option but less intrusive. It consists in writing a library that re-implements all the functions from the API and simply calls the real API functions inside each of them. As in the previous solution, each call to the real API function is surrounded by two tracepoints to collect the beginning and end times of the call. This new library is simply preloaded in the system, with the *LD_PRELOAD* environment variable, and intercepts all the calls to the library used to program the GPU. This technique was already used for HSA (Margheritta, 2017) and OpenCL (Couturier and Dagenais, 2015).

Being able to collect all the function calls related to the GPU is important but not sufficient. Indeed, when a node offloads some tasks to the GPU, the CPU thread that enqueues the work to the GPU queue usually continues directly and does not wait for the GPU to finish the queued work. In this way, the majority of the GPU kernels and memory copies performed by the GPU are asynchronous, and the API tracing will not help to get information about them. As seen in the next subsection, the API tracing should be supplemented with a specific profiling of the GPU.

GPU profiling

Tracing the API can effectively give insights about the GPU activity. However, the timestamps collected are only meaningful from a CPU point of view. In order to have a complete understanding of the execution, we need to get insight into all the asynchronous operations. These operations represent the real work performed by the GPU.

Asynchronous events can be of three types :

1. GPU kernels which represent the computation performed by the GPU
2. Memory copies, representing the memory transfers between the devices (CPU, GPU)
3. Barriers which constitute synchronization points between the device (GPU) and the host (CPU)

In order to collect the beginning and end times for each asynchronous event, we need to use profiling functions offered by the GPU platform. Each platform offers a varying number of options at different abstraction levels.

— High level : In this case, two options exist. The first one consists in registering a callback which is called after the completion of each asynchronous operation. The start and end timestamps are saved by the GPU and can be retrieved inside the callback function. Registering a callback is often available with external profiling libraries like CUPTI or directly possible in the GPU API like OpenCL. The second option consists in a profiling function that directly returns the timestamps for all the GPU kernels or memory copies executed within an application. This method usually requires to enhance the original behavior of the GPU queue with profiling capabilities, in order to save the information about all the operations executed.

— Low level : Sometimes, collecting GPU information can also be performed at a lower level like the HSA level. For that, low level profiling functions are used and neither callbacks nor profiling capabilities of the queue are required. For example, this is possible inside the *HC (Heterogeneous Compute)* library developed by AMD.

Additional information

The goal of profiling a dataflow model is to understand the execution of the application but also to see the interactions between the different devices and to insure that all the hardware resources are used efficiently. For this purpose, two additional information can be useful and are detailed here.

Linux Kernel tracing Even if some operations can offload work to the GPU and benefit from an acceleration, the global execution of the graph is managed by the CPU, possibly on several cores. Therefore, kernel traces can enhance the userspace traces of the application and help to analyze the parallelism aspect of the computation. This is also helpful to understand the I/O operations performed by the application. With the proposed method, it is possible to collect kernel traces and display them jointly with all the events from the userspace level. Two major views exist to display the kernel traces.

— Control Flow View : It shows the state (running, waiting for CPU, blocked on I/O, ...) of every thread that was running during a tracing session.

— Resource View : It shows the state and the frequency of every CPU core, as well as the software and hardware interrupts.

The Linux Kernel traces complete our analysis environment with information from a low level layer and help users to identify and understand performance issues.

Performance counters The second element addresses the performance of the GPU. The main purpose of getting support from a GPU is to speed up operations that involve heavy computation. Still today, tracing precisely the execution of a kernel on the GPU is not possible and the only way to insure a good performance for the GPU kernels is by collecting performance counters. They represent hardware metrics that provide information about the execution of a kernel on the GPU. For example, we can collect counters representing the average number of vector or scalar ALU instructions executed per work-item, the number of wavefronts, the total size of data written to or fetched from the GPU memory, or the percentage of fetch, write, atomic and other instructions that hit the data cache. These metrics give an insight into GPU kernels executions and can be helpful to ensure efficient hardware usage.

Usually, the profiling tools from the GPU vendors offer the possibility to get the performance counters for each GPU kernel of an application. In our case, we used and adapted the work of Margheritta (2017) to collect the counters for every kernel invoked by the application. The result consists in a table with the rows representing the kernels and the columns the different metrics.

The analysis process is relatively easy for applications involving a few kernels. However, in our case, dataflow models usually entail a large number of GPU kernels, especially if we consider several executions of the graph. Therefore, we need a way to connect each kernel and the associated metrics to the node in the graph that invoked the kernel. We also need to have a time reference for each kernel as, during the profiling phase, GPU kernels can be executed several times. All of this was solved and implemented using Babeltrace and Python

scripts.

Combining all this information (dataflow model API, GPU access API, GPU profiling) results in a significant amount of data. After tracing the application and collecting events, we need to post-process the trace. This step is detailed in the next subsection.

4.4.3 Trace Correlation and Analysis

For this step, we are using Babeltrace and its Python bindings to easily read, post-process and write the CTF traces. Two steps are considered.

Building a fully coherent trace

As seen before, most of the computational intensive nodes are offloaded to the GPU, if available. The desired information about the GPU is the beginning and end times of the GPU kernels execution and memory copies. Unfortunately, it can only be collected asynchronously, once the GPU kernel or the memory copy is finished. Therefore, in the instrumentation, we use a temporary event and store the beginning and end timestamps of the GPU operation inside it. The real timestamp of this temporary event is not meaningful regarding the execution of the dataflow model, as it represents the moment when the GPU information was collected. From it, two new events are created during the post-processing phase. One corresponds to the beginning of the operation and the second to the end. Finally, we need to sort all the events according to their timestamp, since the time of a CTF trace should always increase monotonically. The pseudocode of the reordering algorithm is shown in Algorithm 1

A second concern is to insure that all the GPU-related events are coherent with the CPU-related events. Indeed, the same clock should be used. As LTTng is using the Linux Clock Monotonic, we insure that all the events in the trace are converted to this time base. Depending on the library used to program the GPU, some synchronization work might be required. With OpenCL, for example, the timestamps returned by the profiling functions have no specific reference and cannot be converted into a CPU time. For this reason, we need a synchronization method to connect GPU and CPU events. When we are profiling OpenCL on the CPU for the API calls, and on the GPU for the kernel timestamps, we can get the information described in Figure 4.3. We can solve the synchronization problem by using this information and applying techniques like the Convex Hull algorithm presented by Poirier et al. (2010), or the more efficient version of Jabbarifar (2013).

Algorithm 1 Asynchronous events reordering algorithm

```
1: events : a list containing all the events of the trace
2: procedure ReorderAsyncEvents(events)
3:     for event in events do
4:         if event.type == gpu_kernel or event.type == memory_copy or event.type == barrier then
5:             start_event ← new Event
6:             start_event.timestamp ← event.begin_time
7:             end_event ← new Event
8:             end_event.timestamp ← event.end_time
9:             for field in event.fields do
10:                start_event[field] ← event[field]
11:                end_event[field] ← event[field]
12:            end for
13:            Add start_event to events
14:            Add end_event to events
15:            Remove event from events
16:        end if
17:    end for
18:    Sort events
19: end procedure
```

Matching GPU related events with the dataflow graph

The second post-processing step aims at relating the GPU operations with the dataflow model. The computation approach of a GPU is that GPU kernels are enqueued into one or several queues and then the hardware schedules and executes the kernels from the queues. Moreover, a dataflow model usually contains many nodes and each can offload work to the GPU. Therefore, it is important to match the GPU kernels with the nodes in the dataflow graph. This can be implemented as a post-processing task once all the events have been collected. One limitation of this method is the supposition that a unique queue is used to feed work to the GPU. It can be decomposed into two parts :

1. First we match all the function calls that enqueue kernels inside the GPU queue (encircled in red in figure 4.4) with the nodes in the graph (bottom line in figure 4.4). By using the beginning and end timestamps of the nodes in the graph, we can determine for each function call that launches a GPU kernel, the current node being executed at this moment (the red arrows in figure 4.4).
2. The second part is matching the function call that enqueues kernels into the GPU queue and the real execution of the GPU kernel. For this purpose, we assumed that only one queue was used, and therefore the order of the API calls and the order of the GPU kernel executions are the same. Figure 4.4 shows both order.

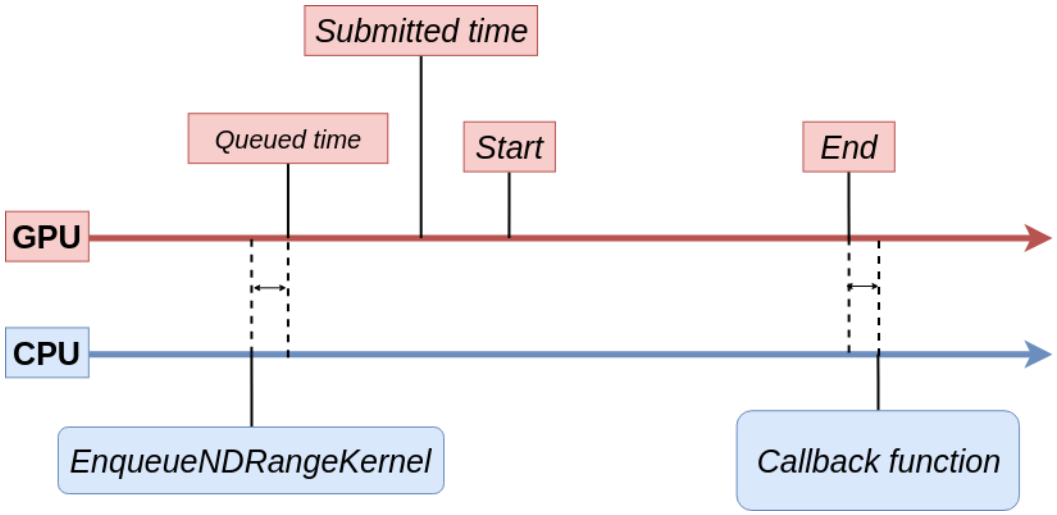


Figure 4.3 OpenCL profiling : Several events happen on both devices and the corresponding timestamps can be collected and used for synchronization

Finally, by combining the two parts, we are able to match all the kernels executed on the GPU with their corresponding node in the dataflow graph. After these two post-processing steps, the trace is coherent and its analysis and visualization can start. The pseudocode for the matching algorithm is available in Algorithm 2.

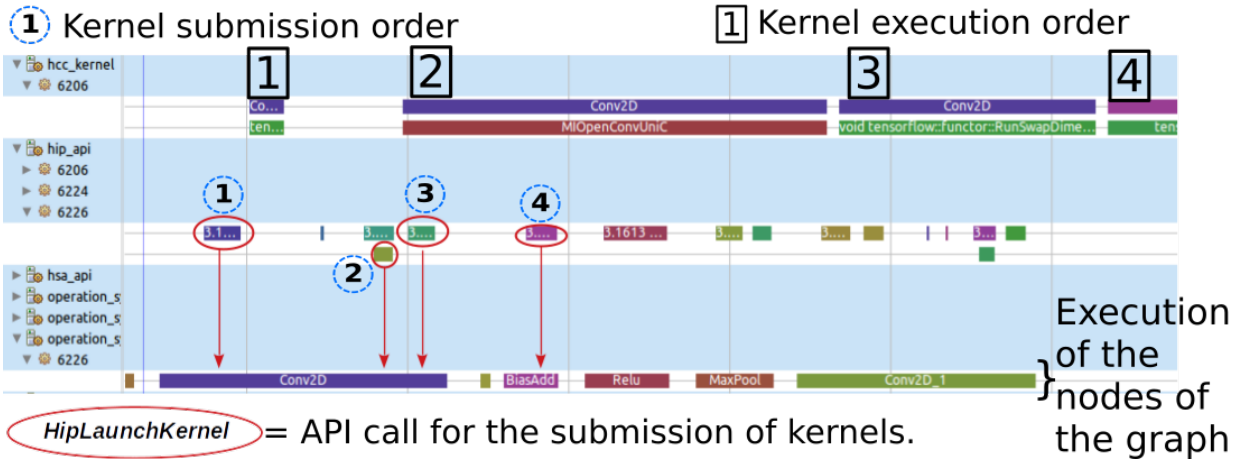


Figure 4.4 Example of matching of the GPU kernels with the nodes from the computation graph

4.4.4 Visualization

In order to profile and trace a dataflow application, the first requirement is to collect the necessary information. After, we also need to develop appropriate and helpful visualizations

Algorithm 2 Nodes and GPU kernels matching algorithm

```
1: events : a list containing all the events of the trace
2: procedure MatchNodesWithGPUKernels(events)
3:     curr_gpu_op ← None
4:     curr_launch_cmd ← None
5:     for event in events do
6:         if event.type == gpu_operation_begin then
7:             curr_gpu_op ← event.name
8:         else if event.type == gpu_operation_end then
9:             curr_gpu_op ← None
10:        else if event.type == launch_kernel_begin then
11:            if curr_gpu_op == None then
12:                Error, a launch command requires a current gpu operation
13:            else
14:                curr_launch_cmd.op_name ← curr_gpu_op.name
15:            end if
16:        else if event.type == gpu_kernel_begin then
17:            if curr_launch_cmd == None then
18:                Error, a GPU kernel begin event requires a current launch command
19:            else
20:                event.op_name ← curr_launch_cmd.op_name
21:            end if
22:        end if
23:    end for
24: end procedure
```

to display all the events in the best possible way to help identifying performance issues. For that, we developed three main elements that will be detailed later : a Callstack view, XY charts, and statistics. As the created trace is in the CTF format, Trace Compass is the best choice for the visualization. Indeed, it is a complete interactive visualization tool aiming at performance analysis of systems. Originally, it focused on operating systems analysis but was extended for different kinds of systems. Moreover, as an open-source software, it can be easily enhanced for all kinds of needs. The main way to develop new views is with Java views, but Wininger et al. (2016) and Kouame et al. (2015) introduced a new possibility called *XML Analysis*. This allows the user to rapidly develop declaratively new trace visualization views and to underline the most important parts. In our work we decided to use this option.

Callstack view

The Callstack view is the main part of the visualization and provides an insight into the execution of the dataflow application. It shows the state of many elements that are grouped into categories corresponding to the different characteristics of the dataflow model. Moreover, it can display nested calls or states by showing several lines. For example, one iteration of the dataflow model corresponds to feeding one batch of data to the graph and retrieving the result from the output nodes. This is a crucial information that should appear when visualizing the trace. Here are the main elements shown in the Callstack view :

— The execution of each node in the dataflow graph with the distinction between the nodes assigned to the CPU and the ones processed by the GPU.
— The beginning and end of all the calls to the functions of the libraries used to program the GPU (CUDA, HIP, HSA, OpenCL, ...)
— GPU related operations : kernels, asynchronous memory copies or synchronization barriers
— Information about the scheduling of all the operations in the graph.

With this information, the user is directly aware of the matching between the nodes of the computation graph and the physical units. Moreover, this view can be combined with Tensorboard that can graphically display the computation graph and help to understand the Callstack view.

Another important information concerns the execution of the operations on different devices. In the case of a CPU, we can easily identify the number of threads and the CPU cores used for the execution. This information is available in the Callstack view and the Resource view and require Linux Kernel tracing. In the case of an operation offloaded to the GPU, we provide textual information within the Callstack view about the division of the work. The dimension

of the grid and the work groups for each GPU kernel are available. With a GPU, we are not able to have precise information about the GPU cores involved for the execution of a specific GPU kernel. The device manages the mapping between the kernel and the GPU cores by itself. Moreover, all the cores are usually identical and, even if some optimizations may be possible, they have very little impact on the performance compared to memory transfers or an efficient work division (grid dimension, work group dimension).

XY charts

Memory consumption is a key element for a data-centric model and XY charts are well adapted for that. When too much memory is required by an application, TensorFlow simply returns an error. It is therefore essential to know the memory consumption of each node of the graph. By combining a XY chart representing the memory consumption with the Callstack view described before, the user can directly spot the nodes responsible. Several XY charts can be developed, depending on the analysis performed on the dataflow model.

In our case, with TensorFlow, a first one shows the memory allocations for each device. As explained before, only following the memory allocations performed by TensorFlow might not be sufficient, especially for the GPU, as it usually allocates all the available memory at the application initialization time. Thus, an additional set of three views display more detailed information about the *BFC allocator* : one for the global allocator statistics, one for the chunks usage and one for the bins statistics.

For memory transfers, a XY chart is also more appropriate and displays the information in a better way, as compared to a Callstack view. Therefore, a specific view displaying all the asynchronous memory transfers was also developed.

Finally, when using TensorFlow in a distributed mode, a critical part is the communication between the machines. As most of the communication is for tensor exchanges, knowing the size of these tensors is important. Before being sent over the network, the tensor values are encoded and, by adding a single tracepoint in the encoding function, we collect the size of every tensor sent to another machine. A XY chart representing these sizes helps the user to directly identify the expensive tensor transfers between machines.

Statistics

Another possibility to show useful information about the execution of a dataflow model is to compute some statistics. This can help to quickly determine which operations in the graph took the longest time for their execution. It also allows a user to compare the execution time of

a node over several executions of the graph. Obviously, in order to find the longest operations in the graph, we need to take into account the work that each node might have offloaded to the GPU. For that, we can benefit from the second post-processing step described earlier, whose goal is to associate each GPU kernel with the corresponding computation node in the dataflow graph. Apart from the duration of the execution of each node, another interesting information is the latency of the nodes in the dataflow model. Within the computation graph, a node can start being executed when all its inputs are ready. Thus, we can compute a metric representing the latency of each node by comparing the moment when the last input of a node has finished and the moment when the node actually started to be executed. In terms of implementation, we used Babeltrace to read the CTF trace, Python to compute the statistics and the results are exported into a CSV file.

4.5 Use cases

After explaining the proposed profiling and tracing method, we evaluated its performance on a few concrete dataflow applications. Our implementation prototype was designed for TensorFlow, so all the examples are TensorFlow applications.

4.5.1 Triplet loss example

The first example is an application for face recognition that uses the triplet loss with TensorFlow, to learn good embedding of faces (Moindrot, 2018). This example is demanding in terms of computations and we demonstrate that our method helps the user to detect two performance limitations. Both are related to an inefficient CPU and GPU usage. We also show that the duration of the application is divided by more than 5 if we apply some optimizations. Now we present each step of the performance analysis process.

If we use the tracing and profiling method described in section 4.4, we obtain the trace shown in Figure 4.5. The view represents the beginning of an execution of the computational graph. As a reminder, one execution of the graph is equivalent to one call of the *run()* function of a TensorFlow *Session* and corresponds to feeding the graph with one batch of data, computing all the nodes in the graph and getting the result. For such dataflow applications, a trace usually shows the same pattern repeated several times. Each instance corresponds to one execution of the dataflow graph. Focusing on execution is a good start for the analysis.

The bottom line represents the state of the graph, if it is being executed or not. The top line corresponds to the GPU kernels executions, and all the lines in the middle represent the execution of some CPU-assigned nodes of the graph. For the sake of clarity, we hide all

other information. We can see that the time between the beginning of the graph execution and the moment when the GPU actually starts to work is quite long. Even more interesting, we can notice that during this time the CPU is processing numerous nodes serially, without parallelism. Indeed, we have several threads but they are processed serially on a unique CPU core.

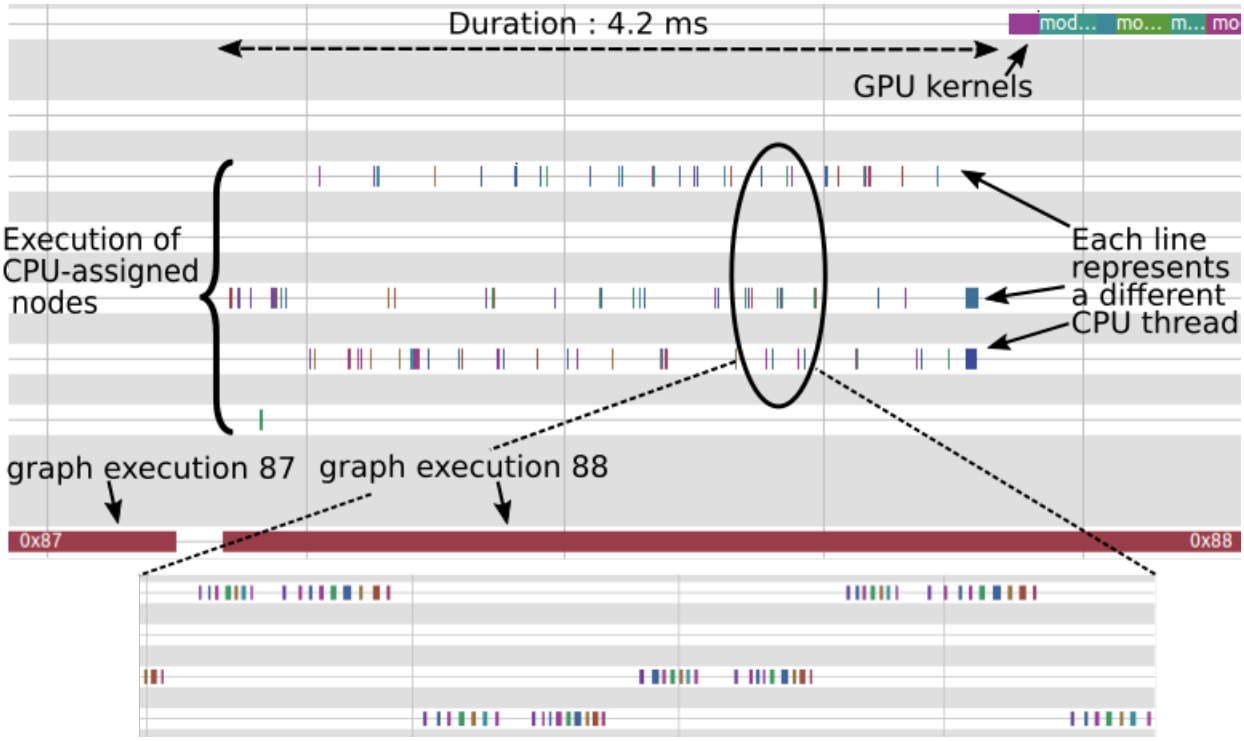


Figure 4.5 Initial application : the beginning of the 88^{th} execution of the graph is shown. The GPU waits for a significant time before starting its job, and the CPU seems to inefficiently process some nodes during this time.

To continue the analysis, we can zoom in to know precisely what are these nodes, as shown in Figure 4.6.

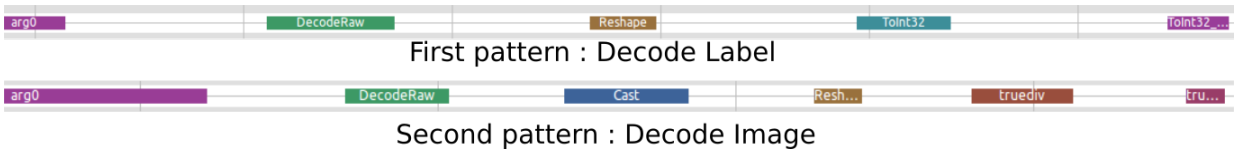


Figure 4.6 CPU-assigned pre-processing operations : The CPU processes many times these two sequences of nodes before the GPU starts.

We can distinguish a pattern, formed out of two sequences of operations, that is repeated many times. The first sequence contains *DecodeRaw*, *Cast*, *Reshape* and *Truediv* nodes and

the second contains *DecodeRaw*, *Reshape* and *Toint32* operations. As they are assigned to the CPU, and are executed before the GPU started to work, we can naturally suspect that they correspond to a pre-processing step on the input data. By looking at the source code, we can notice that these sequences of operations correspond respectively to the *decode_image()* and *decode_label()* functions and that both are passed as an argument to the *map()* function that applies them on images and labels. According to the TensorFlow documentation, the *map()* function offers an argument *num_parallel_calls* that allows to parallelize the processing on several CPU threads. Using this argument can reduce the time needed for this pre-processing step. Figure 4.7 shows the results after having set this argument to 8, the number of CPU logical cores of our machine. The resulting trace proves that the pre-processing has been parallelized on 8 threads, and the duration is now only 2 ms compared to the initial 4.2 ms.

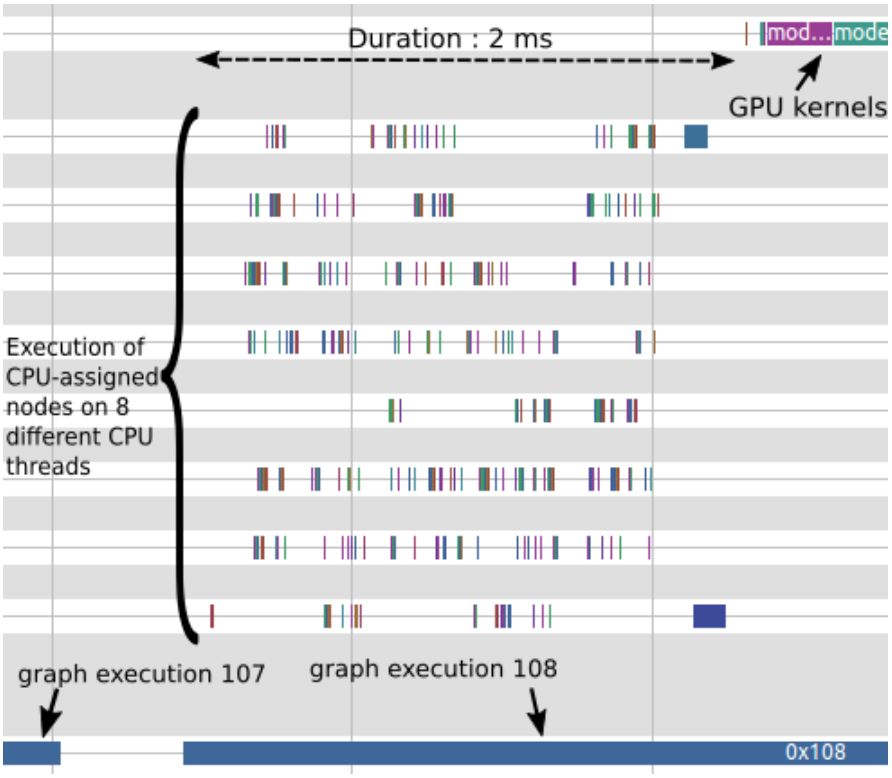


Figure 4.7 Execution with the first optimization : setting *num_parallel_calls* to 8. The processing done on the CPU has been parallelized and is more efficient now. Significant time was saved.

However, we can see that the performance is still not optimal. Indeed, when the GPU starts its job (the top line), the CPU (lines in the middle) is idle. This offers the possibility to implement pipelining, so that the CPU could already pre-process the input data that will be fed into the GPU at the next execution of the graph. Indeed, nodes of the graph that

are entirely executed on the CPU can naturally process data while the GPU is used by the computational intensive nodes. In this way, the data may be almost directly ready for the GPU for each execution of the graph.

TensorFlow enables this with the *prefetch()* function that can be applied to the dataset. Setting its argument to *1* insures that one batch of data is always ready to be fed to the GPU. Figure 4.8 presents the trace with the `map` and `prefetch` optimizations. It is now clear that some nodes are executed on the CPU while the GPU is processing other nodes. Obviously, the duration between the beginning of the session and the moment when the GPU starts working has significantly decreased to 790 μs which corresponds to an improvement by more than **5X**.

Figure 4.8 Execution with the two optimizations : setting *num_parallel_calls* to 8 and *prefetch* to 1. In addition to the parallelization of CPU work, pipelining has been added and increases again the performance.

As a training process with TensorFlow is made of many executions of the graph, the gain in time can be important. If we consider 500000 executions of the graph, we can quickly compute the time spent for pre-processing the data and estimate the gain :

— Original program : $500000 \times 0.0042 = 2100$ seconds $\Rightarrow$ 35 minutes

— Optimized program : $500000 \times 0.000790 = 395$ seconds $\Rightarrow$ 6 minutes 35 seconds

It is clear that the optimized program uses the available hardware much more efficiently, which saves time.

4.5.2 Compute-bound example

As a second use case, we analyze the execution of a convolutional neural network, a very popular model when dealing with images. Training this kind of model requires a large number of iterations and is time consuming. We demonstrate that our method can help to reduce by half the duration of this application.

Figure 4.9 shows the trace obtained with the proposed method. We notice here that the GPU seems to be used all the time. This behavior is desired when a CPU gets support from a GPU, since it indicates that the application fully benefits from the computing power of the GPU. In this case, we are clearly compute-bound, which reduces the improvement possibilities.

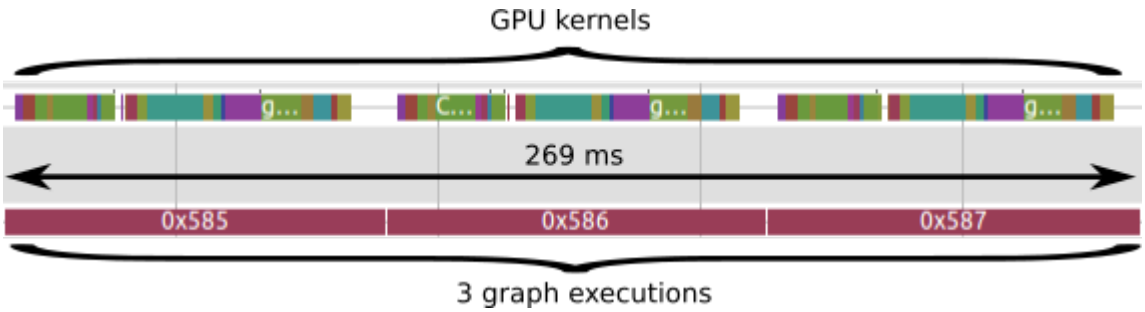


Figure 4.9 Initial execution with the convolution and maxpool operations. The GPU seems to be in use all the time.

However, we can still go deeper and use one of the proposed analysis scripts that compute statistics about the traces. One script returns statistics about the most demanding GPU kernels in terms of duration. The results, presented in Table 4.1 show the mean duration for each kernel, as well as the number of times they are executed during the whole trace. As expected, GPU kernels related to convolutions took the most time. However, the *Maxpool* nodes are also noticeable in terms of duration and frequency.

Since we know that we are compute-bound, we need to reduce the amount of computation to improve the execution of the dataflow graph. We thus target the most demanding nodes. In a paper called *Striving for simplicity : The All Convolutional Net*, Springenberg et al. (2014) explained that under some conditions we can replace a `Convolution+Maxpool` layer by a `Strided Convolution`, without affecting the learning capabilities of the model. Using a strided convolution insures that the shape of the tensor after the convolution is the same as the shape of the tensor after the convolution and maxpool operations. The authors explained that no difference in final accuracy can be perceived for many image recognition benchmarks. Therefore, it is worth considering this optimization in a compute-bound situation.

The Figure 4.10 shows the evolution of the loss and the accuracy for both cases (`convolution`

Table 4.1 Most demanding nodes of the graph. Several of them are related to the MaxPool operations.

Rank	Kernels	Mean duration (ns)	Occur.	Std. dev. (ns)
1	ApplyAdam	1440119	42	238911
2	ReluGrad	1281214	14	2833
3	**MaxPoolGrad**	**986000**	**14**	**4520**
4	BiasAdd	982400	15	208916
5	MatMul	916267	15	8218
6	Relu	860133	15	3403
7	Conv2D_1	849733	45	1189160
8	MatMul_1	747714	14	7639
9	**MaxPool_1_grad**	**586357**	**14**	**29274**
10	Conv2D_1_BackpropFilter	574943	70	1140131
11	**MaxPool**	**547533**	**15**	**45908**
12	BiasAddGrad	521643	14	22292
13	Conv2DBackpropFilter	518786	28	514392
14	BiasAdd_1	446467	15	1928
15	Relu_1	429800	15	3390
16	BiasAdd_1_grad	272500	14	14912
17	**MaxPool_1**	**272500**	**14**	**14913**
...	...	...	...	
62	mul_1	2000	0	14

`+ maxpool` and `strided convolution`) when using a basic convolutional neural network to classify the MNIST digits. As we can see, the learning capabilities of the model are almost not affected.

Figure 4.11 shows the trace at the exact same moment if we replace the convolution and maxpool nodes with a single strided convolution node. Previously, three executions of the graph took 269 ms, compared to 131 ms now, which represents a **2X** benefit. As for the first example, we can do a quick estimation of the gain in terms of duration. If we consider 10000 executions of the graph, we get the following :

— Original Convolution and Maxpool : $10000 \times 0.269 \div 3 = 897$ seconds $\Rightarrow$ 15 minutes

— Strided convolution : $10000 \times 0.131 \div 3 = 437$ seconds $\Rightarrow$ 7 minutes 12 seconds

In spite of a very small accuracy reduction, the time benefit is significant and this optimization can be suitable in many cases. This example concerns a training phase executed on a computer, but we can imagine an inference phase on a mobile device with lower computation resources, where a compute-bound situation is likely. In such a case, the model is expected to make predictions with a very low latency, and considering this kind of optimization may help. The activation functions of the network may present a similar case. Some of them are

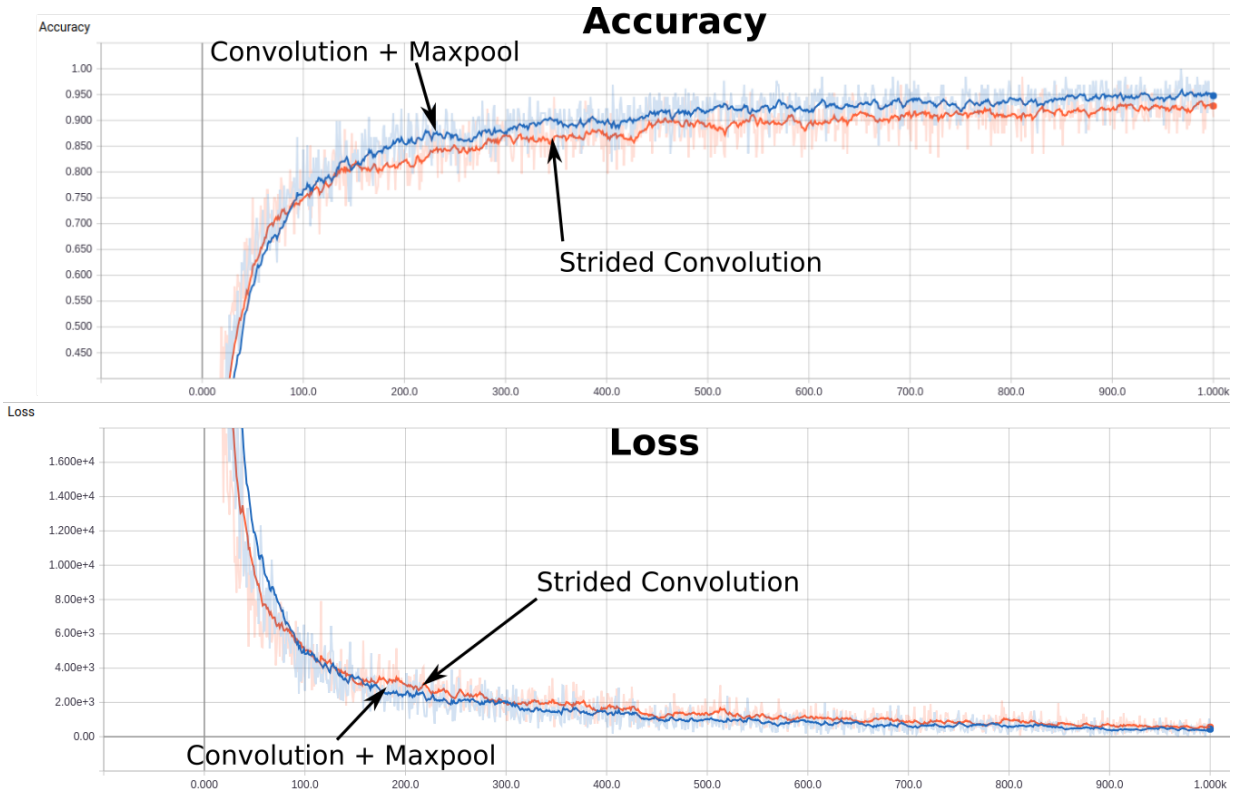


Figure 4.10 Evolution of the accuracy and the loss after 1000 iterations. The accuracy loss caused by the use of strided convolutions with no maxpool layer is subtle.

more demanding in terms of computation, like *"tanh"*. With the proposed method, we can easily understand the limiting elements and then possibly consider alternatives like *"ReLu"*, if we are compute-bound.

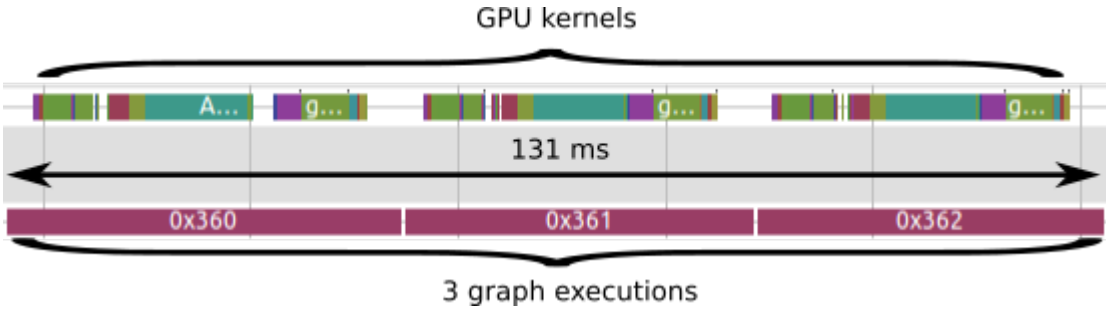


Figure 4.11 Execution with the strided convolution. The GPU is still used all the time, but the duration of 3 executions of the graph has been reduced.

4.5.3 Distributed Dataflow graph

With this work, we also support dataflow models distributed on several machines. It corresponds to the distributed learning option in TensorFlow. This example demonstrates the efficiency of our method to provide useful information about distributed dataflow applications. Information about the partition of the graph on the available machines is particularly valuable. We show that the user benefits a lot from our work in order to optimize an application.

In this use case, we used a convolutional neural network that is distributed on two machines. This technique is named *in-model parallelism*. The second possibility, named *between-model parallelism*, is not explored here but consists in replicating the same graph on several machines to learn in parallel. Finding the best partition of the graph for the device assignment is a difficult problem and there is currently no automatic way for that. Mayer et al. (2017) addressed this difficult problem by proposing and evaluating different strategies. Mirhoseini et al. (2017) proposed an algorithm, based on machine learning techniques, to assign each node of the graph to a specific device.

In our case, the model contains 2 convolutional layers and 1 fully connected layer. We first decided to divide the graph as follows :

— The first convolutional layer is assigned to the first machine.

— The second convolutional layer and the fully connected layer are assigned to the second machine.

For this example, we focus on the forward pass and the effects of the machine boundary between the first and second convolutional layers.

As explained in section 4.4, the tracing and profiling happen on both machines, and the post-processed traces are gathered in one machine for the analysis and visualization step.

Linux Kernel traces are collected at the same time and are necessary in order to synchronize the traces. The result is shown in Figure 4.12.

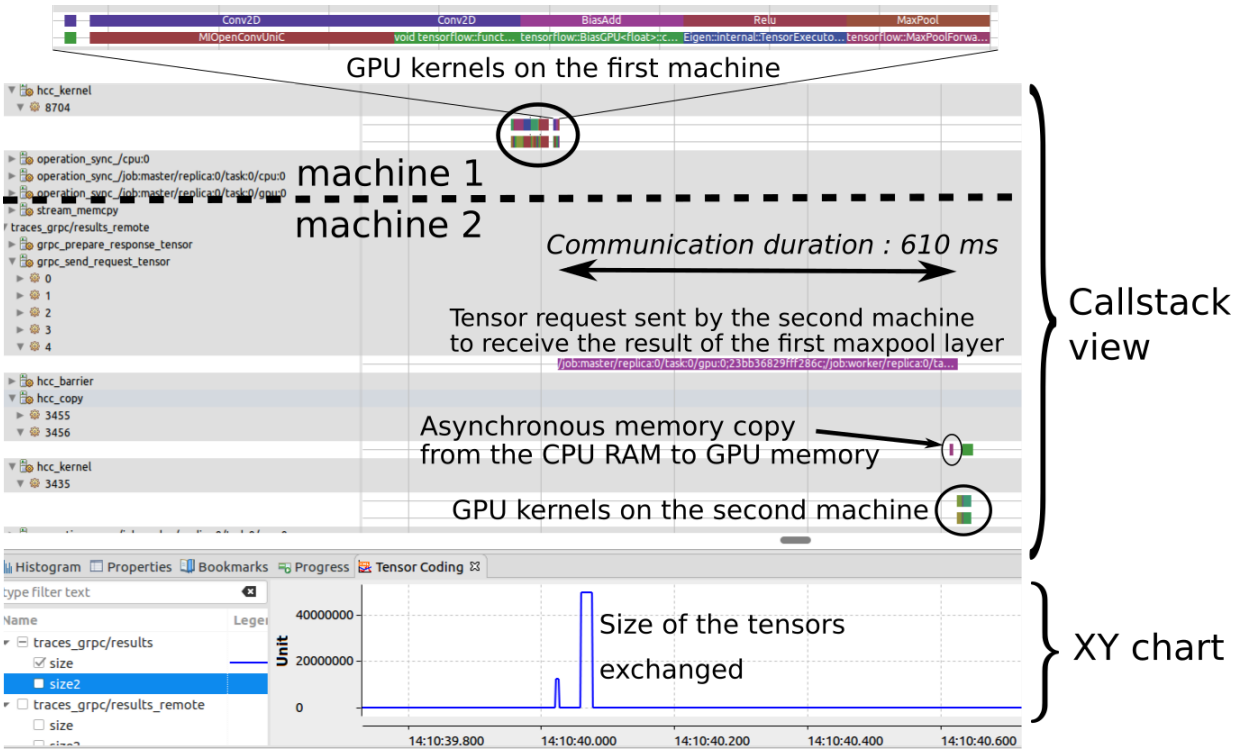


Figure 4.12 Distributed execution of a CNN on 2 machines - first experiment. Machine 1 executes the first convolutional layer and the machine 2 asks for the result. Once received, machine 2 can compute the second convolutional layer and the dense layer. The XY chart at the bottom shows the size of the tensors exchanged between the machines.

The view on top corresponds to the Callstack view of an execution of the graph. We can see the GPU kernels executed on the first machine, which correspond to the first convolutional layer (2D convolution and maxpool nodes). In the view, the execution of the kernels on the GPU always involves two lines, unless we are using TensorFlow with CUDA (like in the two previous examples). The purpose of the top line is to show the real names of the kernels and the line below presents the nodes of the graph responsible for the execution of the GPU kernel. The long line in the middle represents a wait time for the second machine. It starts when the second machine sends the request to the first computer for the result of the first maxpool layer, and ends when the result has arrived on the second computer and can be used as the input for the second convolutional layer.

The XY chart under the Callstack corresponds to the tensor encoding. Before being sent over the network, tensor values are encoded and the line represents the time spent for this,

as well as the amount of data that is sent. The encircled operations on the right, prove that the second machine actually received the data. They correspond to asynchronous memory copies from the host (CPU RAM) to the device (GPU memory) on the second machine. The amount of data sent by the first machine corresponds to the amount of data copied into GPU memory on the second machine : 51380224 bytes. Moreover, we can compute the same result with the shape of the tensor after the first maxpool operation and the batch size : $2048 \times 14 \times 14 \times 32 \times 4 = 51380224$. Around 600 ms is spent to transfer the result between the machines. As the time required to send the tensor represents a large percentage of the execution time (around 2 s), we decided to change the partition of the graph.

In a second experiment, both convolutional layers are assigned to the first machine, and the fully connected layer to the second machine, as shown in Figure 4.13. Again the application

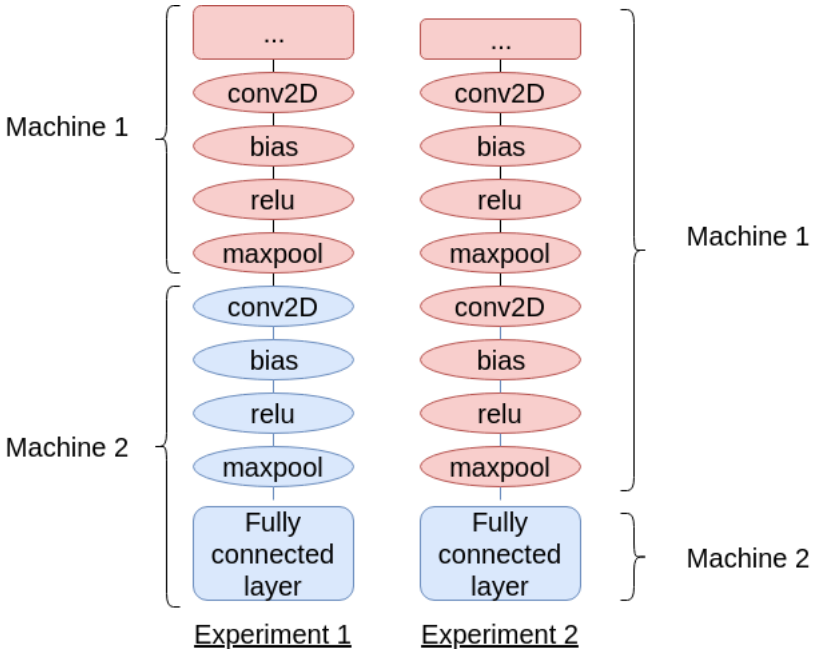


Figure 4.13 Division of the graph for both experiments.

is traced and profiled and the result is compared with the first experiment, shown in Figure 4.14. We can recognize a similar pattern as before, but also note that the time spent for the communication has been reduced to 273.8 ms. Obviously, this is related to the size of the data exchanged between the machines. Applying a second maxpool operation decreases the size of the tensor, which results in a total size of : $2048 \times 7 \times 7 \times 4 = 25690112$ bytes.

Knowing that, we can conclude that the second way of splitting the graph may be more suitable, as the duration of the communication decreased. Since the dataflow model is supposed

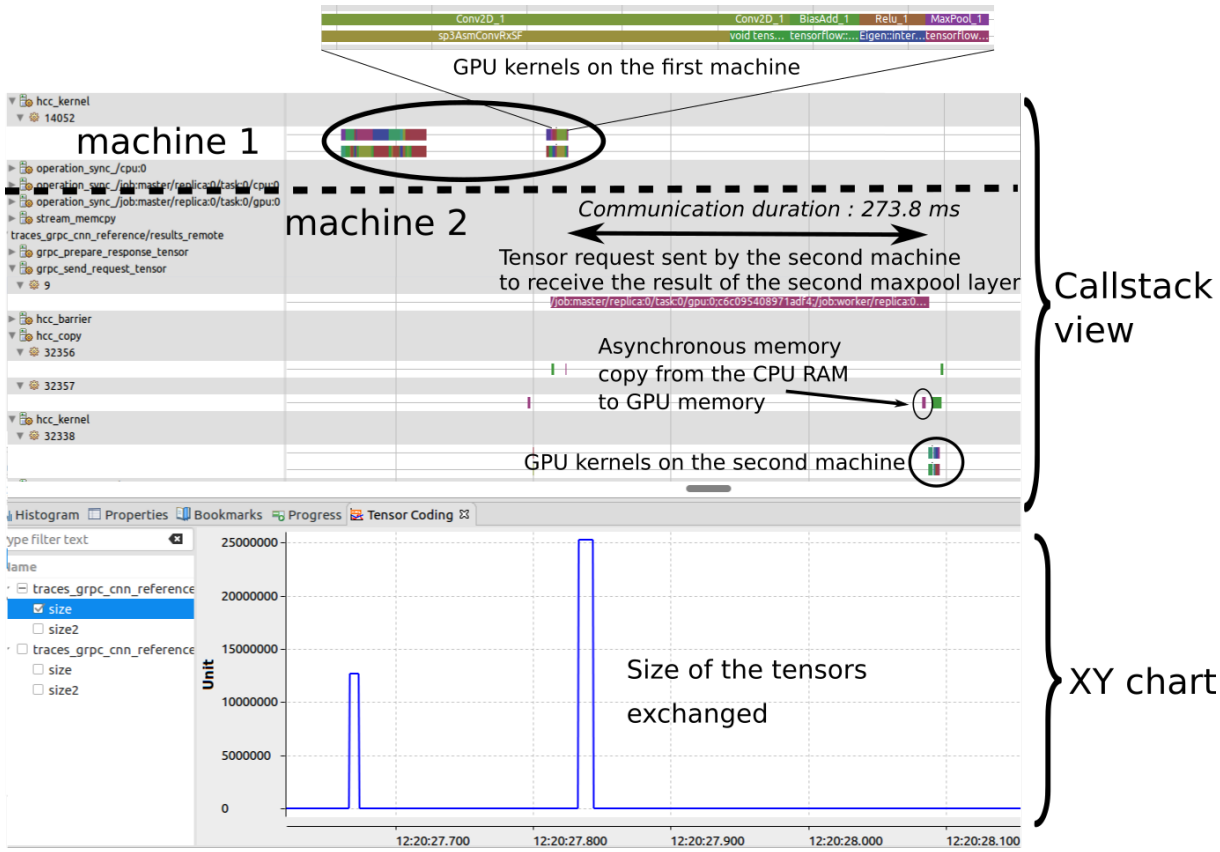


Figure 4.14 Distributed execution of a CNN on 2 machines - second experiment. This time, machine 1 executes the two convolutional layers and machine 2 asks for the result. Once received, machine 2 can compute the dense layer. The XY chart at the bottom shows that the size of the tensors exchanged between the machines is reduced compared to the first experiment.

to be executed many times, the gain can be very important regarding the complete training process. Moreover, here the focus is on a single data transfer between the machines and only for the forward pass. In a real optimization process, we should have considered all the data transfers between the machines and also for the backward pass.

Through this example, we showed that the view obtained with the proposed tool is helpful and brings valuable information to the user. The user can see a detailed trace of the execution : where each operation is executed, how much data is transferred, the transfer direction, the time spent in communication, etc.

In our case, the two machines were similar, but this information can be even more valuable if the two or more machines have different characteristics (GPU vs no GPU) and capabilities (number and frequency of the CPU and GPU cores, memory, ...). For this simple example, only two machines were used but the proposed tool can handle large distributed clusters of

machines.

4.5.4 Inference example

Dataflow graphs are usually intended to be executed many times, but there are a few exceptions. For example, an inference step with machine learning model may consist of a single execution of the graph. On mobile devices, for example, we can have a trained model to classify flowers and want to know the name of a flower in one new image. With this example, we demonstrate the utility of low level information in a performance analysis context. Thanks to it, we were able to decrease the duration and the memory consumption of the application.

Figure 4.15 shows the Callstack view obtained if we trace and profile the application. The first thing we notice is that the time spent by the execution of the dataflow graph only represents a fraction of the total execution time. 1.4 s is spent before the execution of the graph.

With the Kernel Memory Usage view of Trace Compass, we can also see the amount of RAM memory used by the application. This is represented by the bottom line in the XY chart and is around 800 MB at the beginning of the graph execution. In the Callstack view, we do not have real information about what is happening before the beginning of the execution of the graph. This is therefore a good example where the Linux Kernel tracing is useful as shown in Figure 4.16. With this information, we see that a thread was processing and made several system calls (in blue).

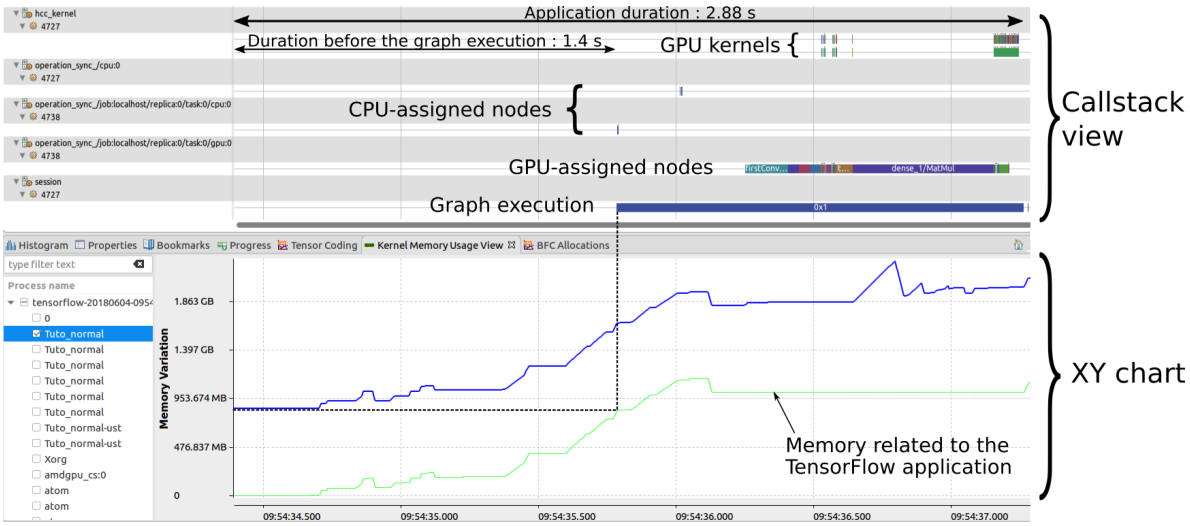


Figure 4.15 Initial Callstack and Linux Kernel memory usage views for an inference step. A long time is spent before the beginning of the graph execution and around 800 MB of RAM memory is used by the TensorFlow application.

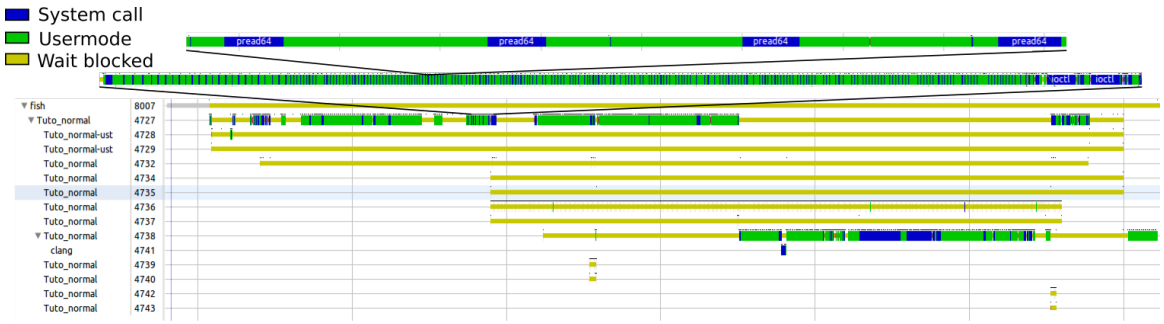


Figure 4.16 Kernel traces of the inference step. We can detect a lot of *pread* system calls. They correspond to the loading of the model file.

If we zoom in, we note that many of these are *pread* calls, which suggests that the application spends a lot of time reading a file. We know that the application has to load the trained model and the learned weights before being able to make predictions. Looking at the source code, we can find a *LoadGraph()* function. As the profiling environment is very flexible, it is easy to quickly instrument this function and to display the result in the Callstack view, together with all the other elements. The instrumentation of *LoadGraph()* is separated in two :

— Reading binary : the first small rectangle (red) in Figure 4.17

— Creating a new session : the largest rectangle (blue) in Figure 4.17

We can also measure that the *LoadGraph()* function call lasted around 400 ms. Figure 4.17 shows the result and proves that this function indeed took some time.

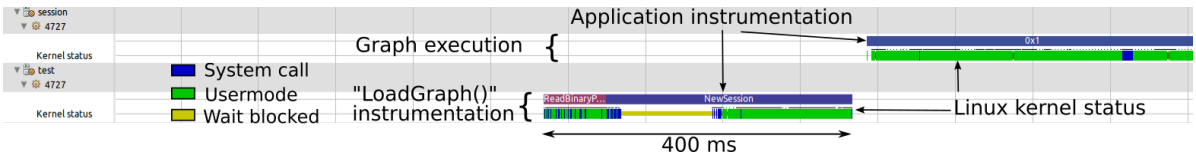


Figure 4.17 Callstack and kernel status, with the static instrumentation of the *LoadGraph* function. We can verify that this function is indeed time consuming.

TensorFlow proposes a complete and well-documented performance guide that introduces a way to load model files more efficiently. This method is a memory mapping technique that maps directly the file into RAM memory. This prevents allocating memory on the heap and then copying bytes from the file into this allocated space. When using this method, we get the result shown in Figure 4.18 and we can note a few elements :

— The total time spent before the beginning of the graph execution has been reduced to 1.28 s (1.4 s previously)

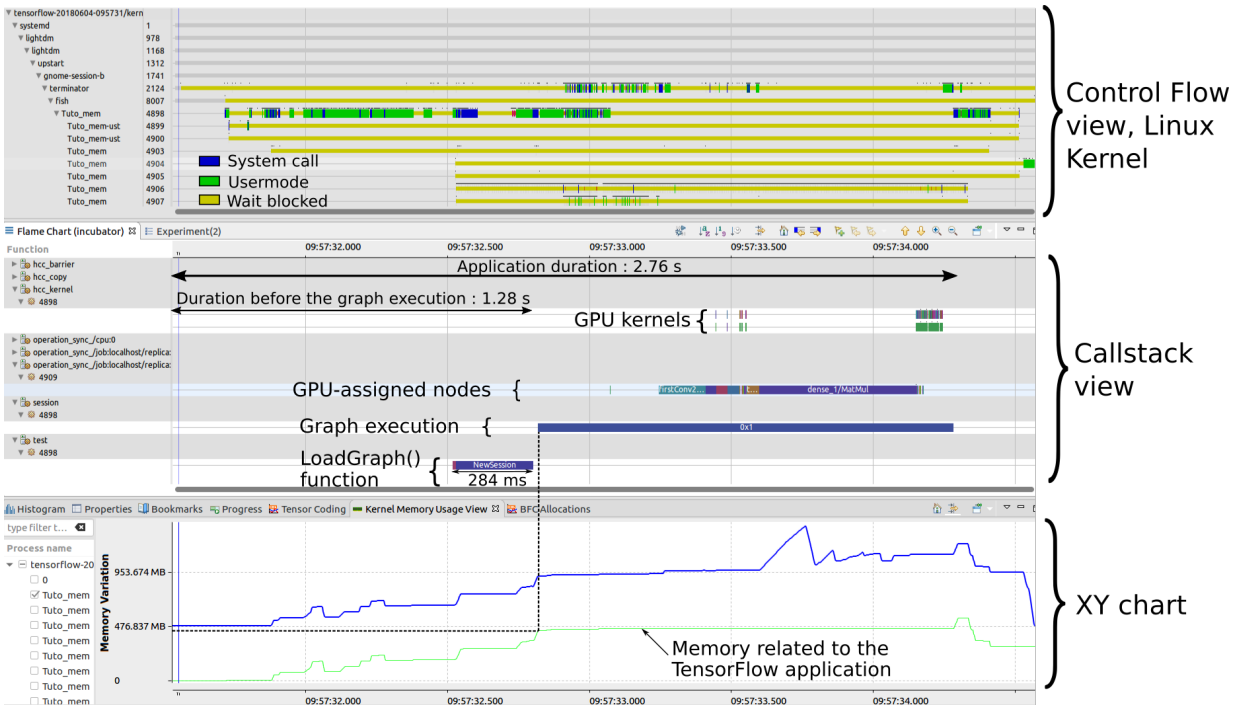


Figure 4.18 Kernel traces, Callstack and Linux Kernel memory usage view of the optimized application. Using the memory mapping technique to load the model file decreases the duration of the *LoadGraph* function. The RAM memory usage is also decreased at the beginning of the graph execution, compared to the first version of the application.

— The duration of the *LoadGraph()* function (reading binary + creating a new session) has decreased to 284 ms (400 ms previously).
— The kernel memory usage of the dataflow application is now around 450 MB, as compared to 800 MB previously.
— As the duration of the graph execution did not change, the total duration of the application gets the same reduction as the loading step : 2.76 s (2.88 s previously)

Through this example, another advantage of our method is pointed out. In addition to a precise profiling of the dataflow graph, we can also benefit from other useful information like the Linux Kernel traces as well as the possibility to easily integrate an external instrumentation of the application.

4.5.5 Memory management example

The last example is about specific situations where the memory management of the dataflow model is problematic. We demonstrate that our method can be applied and lead to a significant reduction of the execution time.

When the majority of the nodes are GPU-accelerated and the dataset can fit into the GPU memory, one large memory copy is probably more efficient than several smaller copies. With TensorFlow, we can imagine an example where the computation graph simply consists of a *square* operation on a matrix (square of the matrix element-wise) and a *reduce_sum* operation that sums all the elements of the matrix. The classic technique consists in feeding a batch of data to the computational graph in TensorFlow at the beginning of each graph execution. Moreover, as the computation nodes of the graph are assigned to the GPU, the data is also copied from the CPU to the GPU. Figure 4.19 shows the profiling and tracing of one execution of the graph.

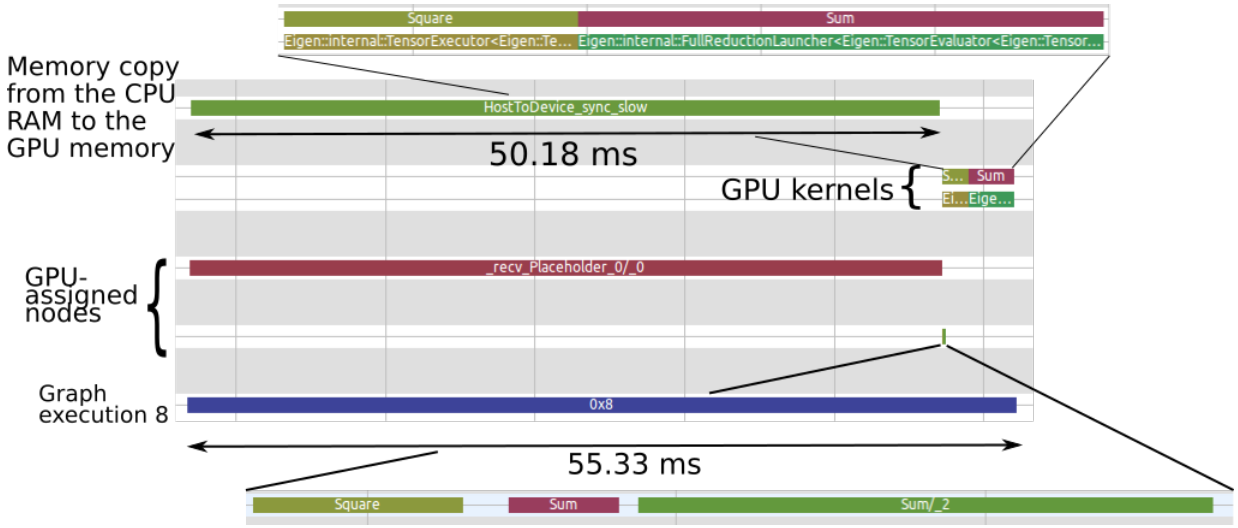


Figure 4.19 Normal execution using feed_dict argument. An important percentage of the execution is dedicated to a large memory transfer from the RAM memory to the GPU memory.

We clearly notice that there is a long memory copy from the Host (CPU) to the Device (GPU), and that it took much more time than the computation part that follows. The *square* and *sum* operations are indeed relatively quickly computed on the GPU and this emphasizes the long time spent in the memory copy. At each execution, a new batch of data is created from the dataset and given to the graph as input. Thus, this phenomenon repeats at each execution of the graph, which means that a very large percentage of the total application time is related to memory copies from the CPU to the GPU.

If the dataset is not too large and can fit into the GPU memory, another option may be more efficient. It consists in copying the whole dataset once to the GPU memory and then adding a new node into the dataflow graph that creates the batches from the whole dataset. In spite of requiring additional time to perform the copy of the dataset once, all the executions of the

graph afterward are much faster, as shown in Figure 4.20.

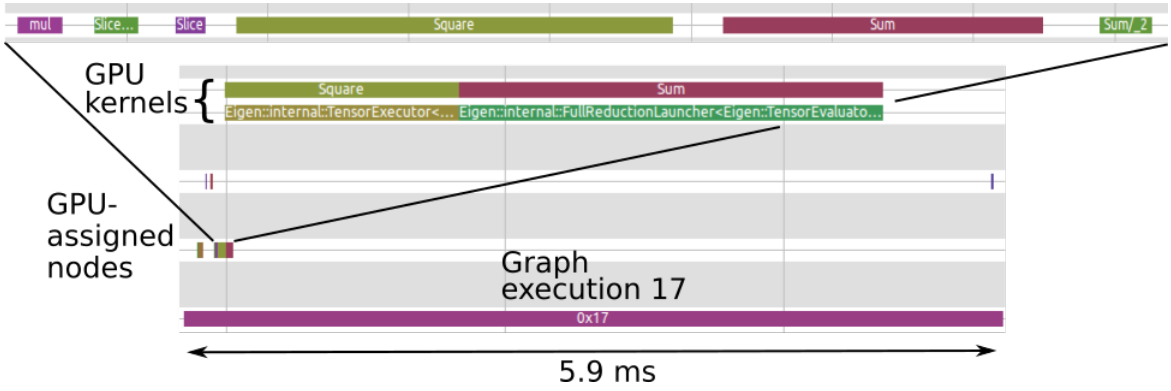


Figure 4.20 After the optimization with a slice operation and a unique large memory copy. Now, the whole dataset is copied to the GPU memory at the application initialization. Consequently, the data batches are already in the GPU memory for all the graph executions. Therefore, each execution is much faster and consists mostly in GPU kernels processing.

Almost all the execution time is now dedicated to the computation part performed by the GPU, and the total time of one execution has decreased by a factor of **10X** compared to the original solution. Obviously, this factor does not take into account the additional large memory copy added at the beginning of the application. However in spite of this, the second option remains advantageous and the speed-up of the whole application simply depends on the number of executions of the graph. The more executions we have, the more beneficial the second solution is. In this example, we showed the utility of this work, when the execution of the dataflow model is hindered by memory transfers.

4.6 Evaluation

In section 4.4, we described the proposed profiling and tracing method for dataflow applications. In section 4.5, we demonstrated its efficiency by applying it to a few examples. The results showed that appropriate analyses and visualizations of the collected traces are valuable. They can help a user to improve the performance of an application. The analysis and optimization processes are usually performed on one or a few executions of the graph, that is why the amount of time saved may not seem significant at first. However, if we run the whole dataflow application, feeding thousands or millions of batches of data to the graph, the time saved can become very important.

When profiling or tracing applications, one key point is the overhead of this operation. It represents a percentage of additional time required when tracing and profiling, as compared

to a normal execution of the application. Tables 4.3, 4.4, 4.5, 4.6, 4.7 and 4.8 show the results for two applications :

— A 2-layer autoencoder used to compress and then reconstruct MNIST digit images
— A convolutional neural network to classify MNIST digits with 2 convolutional layers and 1 fully connected layer

We measured the overhead on the 3 supported platforms for TensorFlow : ROCm, CUDA and SYCL. For each of them, we evaluated the overhead with different tracing and profiling targets.

— Baseline : no tracing or profiling
— Dataflow : collects events related to the dataflow model only
— API : collects functions entry and exit for the GPU API used (HSA, HIP, CUDA, SYCL)
— GPU events : collects GPU related events (kernels, memory copies, usually performed asynchronously)
— Full user : collects dataflow, API and GPU events together
— Full user and kernel : collects dataflow, API, GPU events and all LTTng Linux kernel events together

When using the ROCm platform, our implementation offers three methods to collect GPU related events : by instrumenting the HC library, using interception mechanisms or by analyzing the HC log output. We did the measurements for each of them.

The tracing and profiling process was performed for 500 steps, for both applications, and each was profiled 100 times to compute a mean duration. We chose these number of steps to keep an execution time around a few seconds. Usually, even 10 to 20 steps are sufficient for the performance analysis, as a dataflow model involves the same computation repeatedly performed on new data.

When using TensorFlow with the SYCL backend, a long initialization time appears. This time is constant, whether the application is profiled or not. Therefore, we do not want to consider it when evaluating the overhead. To solve this problem, we run the applications 1000 and 500 times and then we subtract the results. We thus obtain the duration for 500 executions of the graph. As the initialization time is significant, we measure the duration of 25 executions only. We did not run Linux Kernel Tracing in this case since the number of collected events would be too large.

4.6.1 Setup

The hardware and software configurations used for the three cases are described in Table 4.2. TensorFlow with ROCm and SYCL run on almost similar machines. They have the same amount of RAM and contain a similar GPU. On the contrary, a less powerful laptop is used for TensorFlow with CUDA. The same operating system is installed on the three machines and the version of TensorFlow differs depending on the GPU platform.

Table 4.2 Hardware and software configuration

	CUDA	**ROCm**	**SYCL**
TensorFlow version	1.6.0	1.0.1	1.6.0-rc0
GPU library	CUDA 9.0	ROCm 1.7	ComputeCpp 0.7.0
OS	Ubuntu 16.04	Ubuntu 16.04	Ubuntu 16.04
CPU	Intel i7-6700HQ	Intel i7-4770	Intel i7-4790
GPU	Nvidia GTX950M	AMD Nano R9	AMD Nano R9
RAM	16 GoDDR3	32 GoDDR3	32 GoDDR3
GPU memory	2 Go	4 Go	4 Go

4.6.2 Overhead analysis and discussion

The tables 4.3, 4.4, 4.5, 4.6, 4.7 and 4.8 show the duration of an application when enabling different tracing and profiling targets. The overhead compared to a baseline case is also computed. Each table corresponds to one application on one specific platform.

First, we notice that the overhead depends on the application profiled, the information collected, the platform used and also the profiling method used to collect the GPU-related events. For each platform and example, the standard deviation of the execution time stays low, which means that the introduced overhead does not vary much if we analyze the application several times. In addition, the convolutional neural network (Tables 4.6, 4.7 and 4.8) adds a lower overhead compared to the autoencoder (Tables 4.3, 4.4 and 4.5). This is explained by the fact that this application is more demanding in terms of computation and involves longer GPU kernels. The execution of the kernels takes a long time and does not change whether we are profiling or not the application. Therefore, in relative terms, the introduced overhead is lower than with the autoencoder application. Obviously, the more information we want to collect, the higher the overhead, whatever platform and application is used.

One noticeable difference between CUDA, ROCm and SYCL is the tracing of the GPU API. Tracing HSA, HIP or OpenCL introduces a very little overhead compared to the CUDA API. That can be explained by the method used, since for HSA, HIP and OpenCL we use a direct

Table 4.3 Benchmark : Autoencoder - ROCm platform

	Mean time		
Baseline (s)	2.89279		
Standard Deviation (s)	0.03541		
Dataflow (s)	3.00744		
Standard Deviation (s)	0.00880		
Overhead (%)	3.96332		
HSA API (s)	2.94353		
Standard Deviation (s)	0.01060		
Overhead (%)	1.75403		
HIP API (s)	2.89744		
Standard Deviation (s)	0.00611		
Overhead (%)	0.16095		
	HC	**Interception**	**HC log**
GPU events (s)	3.06181	3.12895	3.25691
Standard Deviation (s)	0.01230	0.01060	0.02303
Overhead (%)	5.84278	8.16376	12.58710
Full user (s)	3.40731	3.42743	3.64602
Standard Deviation (s)	0.02388	0.02200	0.03131
Overhead (%)	17.78624	18.48200	26.03825
Full user and kernel (s)	3.54567	3.54247	3.94711
Standard Deviation (s)	0.02141	0.02462	0.04787
Overhead (%)	22.56919	22.45873	36.44666

instrumentation of the libraries whereas tracing the CUDA API is based on asynchronous callbacks from CUPTI.

In general, the overhead stays reasonable and does not cause problems. Indeed, the worst case in our benchmark is for the Autoencoder application on CUDA, when everything is profiled even the Linux Kernel. In that case, the overhead is around 45% which represents approximately one additional second compared to the original execution. Thus, as long as the profiling and tracing process stays relatively short, the overhead of our method is acceptable. Usually, we never trace and profile during a longer period for two reasons.

— Collecting events for a few executions of the graph is enough to analyze the performance, as the output trace shows a repeated pattern.

— A longer tracing and profiling session creates a very large trace which is longer to post-process and visualize.

In the section 4.5, we demonstrated the utility of our method by applying it to several TensorFlow applications. We were able to detect performance limitations and then we applied some optimizations. In the subsection 4.6.2, we proved that the overhead introduced by our

Table 4.4 Benchmark : Autoencoder - CUDA

	Mean time
Baseline (s)	2.17631
Standard Deviation (s)	0.07928
Dataflow (s)	2.20101
Standard Deviation (s)	0.08597
Overhead (%)	1.13495
CUDA API (s)	2.99804
Standard Deviation (s)	0.09566
Overhead (%)	37.75800
GPU events (s)	2.89840
Standard Deviation (s)	0.10709
Overhead (%)	33.17962
Full user (s)	3.01081
Standard Deviation (s)	0.07300
Overhead (%)	38.34465
Full user and kernel (s)	3.17335
Standard Deviation (s)	0.02782
Overhead (%)	45.81318

method stays reasonable. Therefore, our work is beneficial for a user and can definitely be exploited in a performance analysis context. So far, we only implemented it for TensorFlow, but other dataflow applications are conceivable. Another limitation concerns the co-processing units. Only GPUs are supported until now.

Table 4.5 Benchmark : Autoencoder - SYCL

	Mean time
Baseline (s)	4.284931
Standard Deviation (s)	0.61585
Dataflow (s)	4.43434
Standard Deviation (s)	0.80273
Overhead (%)	3.48670
OpenCL API (s)	4.81206
Standard Deviation (s)	0.74740
Overhead (%)	12.3020
GPU events (s)	4.45832
Standard Deviation (s)	0.45399
Overhead (%)	4.04637
Full user (s)	4.98725
Standard Deviation (s)	0.70198
Overhead (%)	16.39053

Table 4.6 Benchmark : Convolutional Neural Network - ROCm platform

	Mean time		
Baseline (s)	6.28211		
Standard Deviation (s)	0.02720		
Dataflow (s)	6.45524		
Standard Deviation (s)	0.05460		
Overhead (%)	2.75592		
HSA API (s)	6.39036		
Standard Deviation (s)	0.05124		
Overhead (%)	1.72311		
HIP API (s)	6.31373		
Standard Deviation (s)	0.01151		
Overhead (%)	0.50327		
	HC	**Interception**	**HC log**
GPU events (s)	6.57057	6.74910	6.98326
Standard Deviation (s)	0.07752	0.02644	0.04407
Overhead (%)	4.59173	7.43362	11.16100
Full user (s)	6.92828	7.20157	7.46502
Standard Deviation (s)	0.02821	0.03647	0.04229
Overhead (%)	10.82588	14.63612	18.82978
Full user and kernel (s)	7.06968	7.37097	7.84885
Standard Deviation (s)	0.02615	0.03128	0.04228
Overhead (%)	12.53670	17.33266	24.93972

Table 4.7 Benchmark : Convolutional Neural Network - CUDA

	Mean time
Baseline (s)	14.52255
Standard Deviation (s)	0.02383
Dataflow (s)	14.63282
Standard Deviation (s)	0.06344
Overhead (%)	0.75930
CUDA API (s)	15.64714
Standard Deviation (s)	0.02333
Overhead (%)	7.74377
GPU events (s)	15.37674
Standard Deviation (s)	0.05243
Overhead (%)	5.88187
Full user (s)	15.83518
Standard Deviation (s)	0.08649
Overhead (%)	9.03860
Full user and kernel (s)	16.16528
Standard Deviation (s)	0.02145
Overhead (%)	16.58716

Table 4.8 Benchmark : Convolutional Neural Network - SYCL

	Mean time
Baseline (s)	23.25167
Standard Deviation (s)	1.59658
Dataflow (s)	23.56750
Standard Deviation (s)	2.34649
Overhead (%)	1.35835
OpenCL API (s)	23.59335
Standard Deviation (s)	3.39059
Overhead (%)	1.46952
GPU events (s)	23.69699
Standard Deviation (s)	1.76584
Overhead (%)	1.91522
Full user (s)	23.87534
Standard Deviation (s)	1.81709
Overhead (%)	2.68227

4.7 Conclusion and future work

In this paper, we presented a profiling and tracing method intended for dataflow applications using GPUs. We detailed each part, collecting the events, post-processing the trace and visualizing the results. We also showed that combining all the information together can bring very useful insight. With that, a user is able to understand the performance of an application and detect bottlenecks, in order to optimize the application and achieve better performance with the same hardware. We demonstrated the efficiency of the method by implementing it for the machine learning library TensorFlow and executing several examples. As we have shown, a distributed execution of a dataflow program on several machines can also benefit from this work. Finally, the flexibility of the method and implementation, as well as its open-source orientation, help to use, adapt, improve and extend the work proposed in this paper.

As future work, it will be interesting to implement the method for other systems. For example, the signal processing domain usually involves dataflow computation models with several co-processing units like GPUs but also DSPs and FPGAs. The post-processing and the visualization steps could also be improved. First, all the processing of the trace is single-threaded and implemented in Python. In order to deal well with traces containing a larger number of events, we could improve these scripts using a more appropriate language like C or C++ and some parallelism. In terms of visualization, new views could also be developed depending on the user needs and the application. Finally, some additional work could be conducted on the GPU and the distribution of the work on all its cores. Currently, no detailed information is available about the different cores involved in the computation of a GPU kernel ; this could lead to new optimizations.

ACKNOWLEDGMENT

The financial support of Ericsson, Ciena, Google, EfficiOS, Prompt and the Natural Sciences and Engineering Research Council of Canada (NSERC) is gratefully acknowledged. We are also grateful to Advanced Micro Devices (AMD) for providing the hardware and software that made this research possible.

CHAPITRE 5 DISCUSSION GÉNÉRALE

Dans ce chapitre, nous revenons sur l'utilisation de la technique proposée, son utilité ainsi que ses performances. Nous abordons également quelques contributions additionnelles faites au cours de ce projet de recherche.

5.1 Cas d'utilisation

Dans l'article du chapitre 4, nous avons présenté cinq cas basés sur TensorFlow afin de démontrer l'efficacité de notre technique. Dans chacun d'entre eux, nous avons tracé et profilé une application afin de collecter des événements. Nous avons ensuite analysé les résultats et cherché les visualisations les plus adaptées afin de comprendre l'exécution de l'application et de mettre en avant d'éventuels problèmes de performance. Dans un second temps, nous nous sommes basés sur la documentation de TensorFlow ainsi que des articles de recherche afin de trouver des solutions aux éventuels problèmes de performance ou limitations. Après ce processus d'optimisation, nous avons tracé et profilé à nouveau l'application afin de vérifier les gains et de comprendre l'impact des modifications apportées. Avec ces cinq exemples, nous avons pu démontrer l'utilité des différentes informations que l'on a choisi de collecter, comme la durée des noyaux de calcul sur le processeur graphique, la durée et la taille des copies de mémoires entre les deux processeurs ou encore les traces du noyau Linux.

Finalement, au travers de ces exemples, nous avons principalement visé les utilisateurs de TensorFlow. Il s'agissait de l'objectif prioritaire pour la méthode proposée. Néanmoins, certaines de ces informations, comme l'ordonnancement des nœuds du graphe ou encore les traces du noyau du système d'exploitation, seraient également bénéfiques aux développeurs de TensorFlow. Au contraire d'un simple utilisateur, ils ont beaucoup plus de liberté et d'opportunités pour modifier le comportement interne de TensorFlow. Par conséquent, nous estimons que notre technique pourrait également être utile dans un contexte d'optimisation de la bibliothèque logicielle TensorFlow elle-même.

5.2 Utilisation

La technique proposée est liée à un environnement de traçage et de profilage très flexible et libre de sources. Nous avons vu que trois plateformes majeures étaient disponibles afin d'utiliser TensorFlow, et que le choix dépendait du type de processeur graphique utilisé. L'avantage de notre implémentation est sa compatibilité avec chacune d'entre elles. L'utilisation se fait

simplement à l'aide de scripts afin d'automatiser le processus d'analyse. Il reste néanmoins possible d'utiliser chaque composant de l'analyse de manière individuelle, ce qui pourrait être utile pour d'éventuels cas spéciaux.

5.3 Performance

La performance de la technique proposée a été évoquée dans la section 4.6 de l'article. Nous avons notamment démontré que le surcoût introduit restait raisonnable dans les différents cas d'utilisation. Ce dernier dépend de la plateforme, de la technique utilisée pour profiler le processeur graphique ainsi que de l'application. Nous avons pu noter qu'une application exigeante en termes de calcul proposera un surcoût plus faible. Cela s'explique par le fait que pendant l'exécution, possiblement longue, des noyaux de calcul sur le processeur graphique, le profilage ou le traçage n'interviennent pas. Il est donc intéressant de comparer notre le surcoût en fonction de la plateforme utilisée et des informations collectées, mais uniquement pour une même application.

Nous pouvons remarquer que dans le pire cas, un surcoût de presque 50 % est introduit pour l'application auto-encodeur. Malgré tout, cela n'est pas problématique pour deux raisons principales. Tout d'abord, il s'agit d'un cas extrême, puisque nous avons décidé de collecter l'ensemble des informations disponibles, incluant les traces du noyau du système d'exploitation. De plus, la faible durée des applications analysées rend un surcoût de presque 50 % concevable. En effet, seules quelques secondes seraient ajoutées en comparaison d'une exécution normale de l'application. Par ailleurs, en imaginant un cas où l'on souhaite conserver un surcoût très faible, il est possible d'activer ou de désactiver précisément chaque point de trace. Ainsi, l'utilisateur pourrait limiter les informations collectées en fonction de ses besoins, afin de conserver un surcoût très faible.

5.4 Contributions additionnelles

Quelques contributions additionnelles ont pu être réalisées au cours de ce projet de recherche. Elles concernent principalement la plateforme ROCm proposée par AMD. Cette dernière comprend une bibliothèque nommée *HIP* pour programmer le processeur graphique. Notre implémentation est basée sur des mécanismes de profilage existants au sein de cette bibliothèque. Cependant, quelques éléments marquant la fin de certaines fonctions de l'interface de programmation manquaient et causaient des problèmes significatifs lors de la phase d'analyse et de visualisation des résultats. Nous avons pu soumettre une correction à cela sur Github.

Par ailleurs, avant d'utiliser `LTTng`, une grande partie du travail avait été implémentée à partir

de mécanismes de profilage proposés par AMD, comme leurs marqueurs (*AMD markers*) pour instrumenter le code source. La visualisation des résultats pouvait s'effectuer dans *CodeXL* mais restait fixe et n'offrait pas de possibilités d'adaptation. Par ailleurs, en s'appuyant sur l'environnement d'analyse d'AMD, il aurait été compliqué de proposer un support pour la plateforme CUDA. Nous avions alors développé un convertisseur permettant de générer des traces CTF à partir des celles au format ATP obtenues avec les outils d'AMD. Cela nous permettait d'analyser et de visualiser les résultats dans Trace Compass. L'étape additionnelle de conversion constituait néanmoins un inconvénient important, en raison de sa durée et son aspect inefficace. Finalement, l'implémentation proposée utilise une approche différente avec notamment *LTTng*, mais ce travail pourrait être réutilisé pour un projet différent.

CHAPITRE 6 CONCLUSION

Pour conclure ce mémoire, nous résumons dans un premier temps nos travaux. Nous expliquons ensuite les limitations de notre approche et enfin nous présentons des améliorations possibles pour un projet de recherche futur.

6.1 Synthèse des travaux

Comme expliqué dans l'article de recherche au chapitre 4, nous proposons une technique de traçage et de profilage destinée aux applications utilisant un modèle flot de données et s'exécutant sur des architectures hétérogènes. L'objectif final est de fournir à l'utilisateur des vues graphiques apportant des informations complètes quant à l'exécution de son application sur le matériel disponible. Cela permet à l'utilisateur de mieux comprendre la performance de son application, de détecter les parties critiques ainsi que d'éventuels éléments limitants. Il peut ainsi s'assurer que l'ensemble du matériel disponible est utilisé efficacement. Les modèles flot de données étant populaires pour programmer des architectures hétérogènes, il essentiel de vérifier que l'application bénéficie effectivement de l'ensemble du matériel et de possibles accélérations.

Afin de pouvoir créer ces différentes vues, il est nécessaire dans un premier temps de collecter de nombreuses informations. Voici un rappel des catégories d'informations que nous avons considérées :

- Informations d'un haut niveau d'abstraction reliées au modèle flot de données lui-même (durée d'exécution et mémoire utilisée pour chaque nœud du graphe, ordonnancement des nœuds, exécution totale du graphe de calcul, ...)
- Analyse de l'activité du processeur graphique : traçage de l'interface de programmation utilisée pour programmer le processeur graphique (OpenCL, CUDA, ...) et récupération des durées des opérations asynchrones effectuées sur le GPU (noyaux de calcul, transferts de mémoire et barrières de synchronisation)
- Traçage du noyau du système d'exploitation (états des fils d'exécution, états des cœurs du processeur central, appels système ...)
- Compteurs de performance pour chaque noyau de calcul exécuté sur le processeur graphique.

Pour cette étape, nous avons décidé d'utiliser *LTTng*, pour sa performance, sa simplicité de prise en main ainsi que ses fonctionnalités. Une instrumentation de différentes librairies a été mise en place. On retrouve notamment la bibliothèque logicielle proposant le modèle flot de

données (TensorFlow dans notre cas), mais aussi celles permettant de programmer le processeur graphique (HIP, HSA, OpenCL et CUDA). Enfin, nous avons également instrumenté certains outils ou fonctions de profilage liés au processeur graphique (HC et OpenCL).

Tel qu'expliqué dans l'article, certaines informations ne sont pas exploitables directement. On retrouve notamment l'ensemble des événements liés aux opérations asynchrones effectuées sur le processeur graphique. De même, de nouvelles informations peuvent être dégagées à partir des événements collectés. Ainsi, une étape de traitement a posteriori des traces est nécessaire. Les résultats étant au format CTF, nous utilisons Babeltrace pour lire et écrire les traces lors de cette étape.

Finalement, il reste donc à présenter les résultats à l'utilisateur. Pour cela nous utilisons différents procédés :

— Vues graphiques dans Trace Compass
— Calculs de statistiques à partir de la trace
— Présentation sous forme de tableau des compteurs de performance

Un avantage important de la méthode et de l'environnement de traçage et de profilage proposés concerne leur flexibilité et leur caractère libre de sources. Les informations présentées précédemment ont été définies après une longue phase de réflexion. Cependant, il ne serait pas difficile pour un utilisateur d'ajouter d'autres informations que ce dernier considérerait comme importantes dans son cas. L'extension serait aisée et rapide en termes d'instrumentation, mais aussi pour les différents scripts de traitement a posteriori des traces, de calcul de statistiques ou encore ceux définissant les visualisations dans Trace Compass.

6.2 Limitations de la solution proposée

Une des limitations de notre solution est liée au caractère très récent de certains travaux. On retrouve par exemple la plateforme ROCm, qui est utilisée pour le support de processeurs graphiques AMD pour TensorFlow. Cette dernière est relativement récente, ses composants sont toujours en développement, et de nombreuses évolutions sont à prévoir. La méthode proposée dans ce travail de recherche se base sur certains composants de ROCm, en particulier pour le profilage du processeur graphique. Par conséquent, un travail de mise à jour de notre implémentation serait nécessaire afin d'assurer son fonctionnement avec les nouvelles versions de ROCm.

Un second élément limitant concerne les instrumentations mises en place. Bien que nécessaires, elles requièrent une recompilation des bibliothèques. Un travail initial de préparation de l'environnement est donc nécessaire avant de pouvoir utiliser notre technique avec Ten-

sorFlow.

Pour le traitement a posteriori des traces ainsi que pour les visualisations dans Trace Compass, nous avons émis la supposition qu'une seule queue était utilisée pour soumettre les noyaux de calcul au processeur graphique. Il est néanmoins possible d'avoir d'autres queues spécifiques pour gérer les copies de données dans chaque direction entre les processeurs. Cette supposition est toujours vraie dans le cas de TensorFlow mais pourrait devenir problématique pour d'autres librairies.

Par ailleurs, le traitement a posteriori des traces et les méthodes de visualisation développées impliquent également une limite sur la taille des traces. En effet, un trop grand nombre d'événements peut être problématique en termes de durée de traitement et de fluidité de visualisation. Malgré tout, dans notre contexte il est relativement rare de vouloir tracer et profiler une application durant une longue période. En effet, chaque exécution sur un jeu de données est relativement courte, mais répétée possiblement des millions de fois avec d'autres données.

Enfin, une dernière limitation concerne le mode distribué de TensorFlow. Notre travail traite le cas où deux machines sont utilisées. Cependant, une situation avec un grand nombre de machines pourrait être problématique en termes de traitement a posteriori des traces et visualisations des résultats.

6.3 Améliorations futures

Différentes améliorations sont envisageables et constituent principalement des réponses aux limitations exposées précédemment.

Tout d'abord, les scripts de post-traitement de la trace pourraient être améliorés. Pour l'instant, ils ont été développés en Python et n'utilisent aucun parallélisme. Par conséquent, leur exécution est lente et peut être problématique lors du traitement d'une trace contenant un très grand nombre d'événements. Il serait envisageable de les implémenter dans un langage offrant de meilleures performances comme le C ou C++ et pouvant utiliser efficacement plusieurs fils d'exécution. De plus, les algorithmes de post-traitements pourraient être perfectionnés afin de traiter le cas où plusieurs queues pourraient soumettre des noyaux de calcul au processeur graphique.

Au niveau des vues développées pour la visualisation des traces, des améliorations sont également possibles. Transformer les vues XML développées en vues Java pourrait permettre de bien mieux gérer les traces ayant un très grand nombre d'événements. La fluidité pourrait être améliorée et des analyses comme le suivi du chemin critique pourraient alors être menées.

Concernant le profilage et traçage de la carte graphique, moins d'informations sont disponibles en comparaison du processeur traditionnel. Par exemple, il n'est pas possible de connaître précisément les cœurs du GPU impliqués pour le calcul d'un noyau. Les différents cœurs sont en général similaires et gérés automatiquement par la carte graphique. L'association entre les cœurs du GPU et les noyaux de calcul n'a pas un grand impact sur la performance, en comparaison de la gestion des transferts de mémoire par exemple. Malgré tout, des optimisations pourraient être envisageables en s'appuyant sur une bonne localité des ressources.

Enfin, il serait intéressant d'implémenter notre technique de profilage et traçage pour une autre bibliothèque logicielle basée sur un modèle flot de données. Le domaine du traitement du signal offre de nombreux cas possibles avec différents travaux basés sur ce genre d'approche.

RÉFÉRENCES

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, et al., "Tensorflow : Large-scale machine learning on heterogeneous distributed systems", *CoRR*, vol. abs/1603.04467, 2016. En ligne : http://arxiv.org/abs/1603.04467

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, et al., "Tensorflow : A system for large-scale machine learning", *CoRR*, vol. abs/1605.08695, 2016. En ligne : http://arxiv.org/abs/1605.08695

M. Abadi, M. Isard, et D. G. Murray, "A computational model for tensorflow (an introduction)", 2017. En ligne : http://dl.acm.org/citation.cfm?doid=3088525.3088527

R. Al-Rfou, G. Alain, A. Almahairi, C. Angermüller, D. Bahdanau, N. Ballas, et al., "Theano : A python framework for fast computation of mathematical expressions", *CoRR*, vol. abs/1605.02688, 2016. En ligne : http://arxiv.org/abs/1605.02688

G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", dans *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, série AFIPS '67 (Spring). New York, NY, USA : ACM, 1967, pp. 483–485. DOI : 10.1145/1465482.1465560. En ligne : http://doi.acm.org/10.1145/1465482.1465560

T. Beattie, "Investigation of the SYCL for OpenCL Programming Model", 2015. En ligne : https://static.ph.ed.ac.uk/dissertations/hpc-msc/2014-2015/InvestigationoftheSYCLforOpenCLProgrammingModel.pdf

J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, et al., "Theano : a CPU and GPU math expression compiler", dans *Proceedings of the Python for Scientific Computing Conference (SciPy)*, Juin 2010, oral Presentation.

E. Bezati, M. Mattavelli, et M. Raulet, "Rvc-cal dataflow implementations of mpeg avc/h.264 cabac decoding", dans *2010 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Oct 2010, pp. 207–213. DOI : 10.1109/DASIP.2010.5706266

S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. Von Platen, M. Mattavelli, et M. Raulet, "OpenDF - A Dataflow Toolset for Reconfigurable Hardware and Multicore

Systems", dans *Multi-Core Computing. MCC 2008. First Swedish Workshop on*, Ronneby, Sweden, Nov. 2008, p. CD. En ligne : https://hal.archives-ouvertes.fr/hal-00340437

T. Blattner, W. Keyrouz, S. S. Bhattacharyya, M. Halem, et M. Brady, "A hybrid task graph scheduler for high performance image processing workflows", *Journal of Signal Processing Systems*, vol. 89, no. 3, pp. 457–467, Dec 2017. DOI : 10.1007/s11265-017-1262-6. En ligne : https://doi.org/10.1007/s11265-017-1262-6

C. Bourrasset, L. Maggiani, J. Sérot, et F. Berry, "Dataflow object detection system for fpga-based smart camera", *IET Circuits, Devices Systems*, vol. 10, no. 4, pp. 280–291, 2016. DOI : 10.1049/iet-cds.2015.0071

——, "Dataflow object detection system for fpga-based smart camera", *IET Circuits, Devices Systems*, vol. 10, no. 4, pp. 280–291, 2016. DOI : 10.1049/iet-cds.2015.0071

J. Boutellier et T. Nyländen, "Design Flow for GPU and Multicore Execution of Dynamic Dataflow Programs", *Journal of Signal Processing Systems*, vol. 89, no. 3, pp. 469–478, 2017. DOI : 10.1007/s11265-017-1260-8

J. Boutellier et T. Nyländen, "Design flow for gpu and multicore execution of dynamic dataflow programs", *Journal of Signal Processing Systems*, vol. 89, no. 3, pp. 469–478, Dec 2017. DOI : 10.1007/s11265-017-1260-8. En ligne : https://doi.org/10.1007/s11265-017-1260-8

J. Boutellier et T. Nylanden, "Programming graphics processing units in the rvc-cal dataflow language", dans *2015 IEEE Workshop on Signal Processing Systems (SiPS)*, Oct 2015, pp. 1–6. DOI : 10.1109/SiPS.2015.7344994

J. Boutellier, J. Wu, H. Huttunen, et S. S. Bhattacharyya, "PRUNE : dynamic and decidable dataflow for signal processing on heterogeneous platforms", *CoRR*, vol. abs/1802.06625, 2018. En ligne : http://arxiv.org/abs/1802.06625

S. C. Brunet, M. Mattavelli, et J. W. Janneck, "Profiling of dataflow programs using post mortem causation traces", dans *2012 IEEE Workshop on Signal Processing Systems*, Oct 2012, pp. 220–225. DOI : 10.1109/SiPS.2012.54

——, "Profiling of dataflow programs using post mortem causation traces", *IEEE Workshop on Signal Processing Systems, SiPS : Design and Implementation*, pp. 220–225, 2012. DOI : 10.1109/SiPS.2012.54

M. Canale, S. Casale-Brunet, E. Bezati, M. Mattavelli, et J. W. Janneck, “Dataflow programs analysis and optimization using model predictive control techniques : An example of bounded buffer scheduling”, dans *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, Oct 2014, pp. 1–6. DOI : 10.1109/SiPS.2014.6986054

M. Canale, ·. S. Casale-Brunet, ·. E. Bezati, ·. M. Mattavelli, ·. J. Janneck, S. Casale-Brunet, E. Bezati, M. Mattavelli, M. M. Ch, et J. Janneck, “Dataflow Programs Analysis and Optimization Using Model Predictive Control Techniques Two Examples of Bounded Buffer Scheduling : Deadlock Avoidance and Deadlock Recovery Strategies”, *Journal of Signal Processing Systems*, vol. 84, pp. 371–381, 2016. DOI : 10.1007/s11265-015-1083-4. En ligne : https://link.springer.com/content/pdf/10.1007{%}2Fs11265-015-1083-4.pdf

A. Carlsson, J. Eker, T. Olsson, et C. Von Platen, “Scalable parallelism using dataflow programming”, *Ericsson Review*, vol. 2, no. 1, pp. 16–21, 2010.

P. Caspi, D. Pilaud, N. Halbwachs, et J. A. Plaice, “Lustre : A declarative language for real-time programming”, dans *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, série POPL ’87. New York, NY, USA : ACM, 1987, pp. 178–188. DOI : 10.1145/41625.41641. En ligne : http://doi.acm.org/10.1145/41625.41641

D. Couturier et M. R. Dagenais, “Lttng clust : A system-wide unified cpu and gpu tracing tool for opencl applications”, *Adv. Soft. Eng.*, vol. 2015, pp. 2 :2–2 :2, Jan. 2015. DOI : 10.1155/2015/940628. En ligne : https://doi.org/10.1155/2015/940628

J. Demšar, T. Curk, A. Erjavec, Črt Gorup, T. Hočevar, M. Milutinovič, et al., “Orange : Data mining toolbox in python”, *Journal of Machine Learning Research*, vol. 14, pp. 2349–2353, 2013. En ligne : http://jmlr.org/papers/v14/demsar13a.html

M. Desnoyers, “Common Trace Format (CTF) Specification (v1.8.2) ,1er éd., (EfficiOS)”, 2014. En ligne : http://git.efficios.com/?p=ctf.git;a=blob_plain;f=common-trace-format-specification.md

M. Desnoyers et M. R. Dagenais, “Lockless multi-core high-throughput buffering scheme for kernel tracing”, *SIGOPS Oper. Syst. Rev.*, vol. 46, no. 3, pp. 65–81, Déc. 2012. DOI : 10.1145/2421648.2421659. En ligne : http://doi.acm.org/10.1145/2421648.2421659

M. Desnoyers et M. R Dagenais, “The LTTng tracer : A low impact performance and behavior monitor for GNU/Linux”, *OLS (Ottawa Linux Symposium)*, 2006.

A. Doumoulakis, R. Keryell, et K. O'brien, "SYCL C++ and OpenCL interoperability experimentation with triSYCL ACM Reference format", vol. 8, 2017. DOI : 10.1145/3078155.3078188. En ligne : http://dx.doi.org/10.1145/3078155.3078188

C. F. Eigler, "Problem Solving With Systemtap", 2018.

J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, et al., "Taming heterogeneity - the ptolemy approach", *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, Jan 2003. DOI : 10.1109/JPROC.2002.805829

J. Eker et J. W. Janneck, "Cal language report : Specification of the cal actor language", 2003.

P.-M. Fournier, M. Desnoyers, et M. R. Dagenais, "Combined tracing of the kernel and applications with LTTng", dans *Proceedings of the 2009 Linux Symposium*, Juil. 2009. En ligne : /static/publications/fournier-combined-tracing-ols2009.pdf

N. Galoppo, N. K. Govindaraju, M. Henson, et D. Manocha, "Lu-gpu : Efficient algorithms for solving dense linear systems on graphics hardware", *ACM/IEEE SC 2005 Conference (SC'05)*, pp. 3–3, 2005.

D. Goddeke, S. H. M. Buijssen, H. Wobker, et S. Turek, "Gpu acceleration of an unmodified parallel finite element navier-stokes solver", dans *2009 International Conference on High Performance Computing Simulation*, June 2009, pp. 12–21. DOI : 10.1109/HPCSIM.2009.5191718

M. Goli, L. Iwanski, et A. Richards, "Accelerated Machine Learning Using TensorFlow and SYCL on OpenCL Devices", dans *Proceedings of the 5th International Workshop on OpenCL*, série IWOCL 2017. New York, NY, USA : ACM, 2017, pp. 8 :1—-8 :4. DOI : 10.1145/3078155.3078160. En ligne : http://doi.acm.org/10.1145/3078155.3078160

D. Goulet, "Unified kernel/user-space efficient linux tracing architecture", Mémoire de maîtrise, École Polytechnique de Montréal, 2012, retrieved from https ://publications.polymtl.ca/842/.

B. Gregg, "strace wow much syscall", 2014. En ligne : http://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html

B. Gregg et J. Mauro, *DTrace : Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*, 1er éd. Upper Saddle River, NJ, USA : Prentice Hall Press, 2011.

B. N. Gregg, “From DTrace to Linux”, 2014, nov. En ligne : http://tracingsummit.org/w/images/9/9d/TracingSummit2014_FromDTraceToLin

N. Halbwachs, P. Caspi, P. Raymond, et D. Pilaud, “The synchronous data flow programming language lustre”, *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep 1991. DOI : 10.1109/5.97300

M. Hentati, Y. Aoudni, J. F. Nezan, et M. Abid, “A hierarchical implementation of hadamard transform using rvc-cal dataflow programming and dynamic partial reconfiguration”, dans *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing*, Oct 2012, pp. 1–7.

M. Hoffmann, A. Lattuada, J. Liagouris, V. Kalavri, D. Dimitrova, S. Wicki, et al., “Snailtrail : Generalizing critical paths for online analysis of distributed dataflows”, dans *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA : USENIX Association, 2018, pp. 95–110. En ligne : https://www.usenix.org/conference/nsdi18/presentation/hoffmann

M. Jabbarifar, “On line trace synchronization for large scale distributed systems”, Thèse de doctorat, École Polytechnique de Montréal, 2013.

J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, et M. Raulet, “Synthesizing hardware from dataflow programs : An MPEG-4 simple profile decoder case study”, dans *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, Washington, United States, Oct. 2008, pp. 287 – 292. DOI : 10.1109/SIPS.2008.4671777. En ligne : https://hal.archives-ouvertes.fr/hal-00336518

W. J. Jörn, D. M. Ian, et B. P. Dave, “Profiling dataflow programs”, *2008 IEEE International Conference on Multimedia and Expo, ICME 2008 - Proceedings*, pp. 1065–1068, 2008. DOI : 10.1109/ICME.2008.4607622

N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, et al., “In-datacenter performance analysis of a tensor processing unit”, *CoRR*, vol. abs/1704.04760, 2017. En ligne : http://arxiv.org/abs/1704.04760

R. Keryell, R. Reyes, et L. Howes, “Khronos sycl for opencl : A tutorial”, dans *Proceedings of the 3rd International Workshop on OpenCL*, série IWOCL ’15. New York, NY, USA : ACM, 2015, pp. 24 :1–24 :1. DOI : 10.1145/2791321.2791345. En ligne : http://doi.acm.org/10.1145/2791321.2791345

D. Kirk, “Nvidia cuda software and gpu parallel computing architecture”, dans *Proceedings of the 6th International Symposium on Memory Management*, série ISMM ’07. New York, NY, USA : ACM, 2007, pp. 103–104. DOI : 10.1145/1296907.1296909. En ligne : http://doi.acm.org/10.1145/1296907.1296909

E. Kohler, R. Morris, B. Chen, J. Jannotti, et M. F. Kaashoek, “The click modular router”, *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Août 2000. DOI : 10.1145/354871.354874. En ligne : http://doi.acm.org/10.1145/354871.354874

D. Komatitsch, G. Erlebacher, D. Göddeke, et D. Michéa, “High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster”, *Journal of Computational Physics*, vol. 229, no. 20, pp. 7692–7714, 2010. DOI : https://doi.org/10.1016/j.jcp.2010.06.024. En ligne : http://www.sciencedirect.com/science/article/pii/S0021999110003396

K. Kouame, N. Ezzati-Jivan, et M. R. Dagenais, “A flexible data-driven approach for execution trace filtering”, dans *2015 IEEE International Congress on Big Data*, June 2015, pp. 698–703. DOI : 10.1109/BigDataCongress.2015.112

A. Krizhevsky, I. Sutskever, et G. E. Hinton, “Imagenet classification with deep convolutional neural networks”, dans *Advances in Neural Information Processing Systems 25 : 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, 2012, pp. 1106–1114. En ligne : http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

G. Kyriazis, “Heterogeneous System Architecture : A Technical Review”, pp. 1–18, 2012.

E. S. Larsen et D. McAllister, “Fast matrix multiplies using graphics hardware”, dans *Supercomputing, ACM/IEEE 2001 Conference*, Nov 2001, pp. 43–43. DOI : 10.1145/582034.582089

J. Lawrence, J. Malmsten, A. Rybka, D. A. Sabol, et K. Triplin, “Comparing tensorflow deep learning performance using cpus , gpus , local pcs and cloud”, 2017.

D. Lea, “A Memory Allocator”, 1996. En ligne : http://g.oswego.edu/dl/html/malloc.html

E. A. Lee, “Consistency in dataflow graphs”, dans *Proceedings of the International Conference on Application Specific Array Processors*, Sep 1991, pp. 355–369. DOI :

10.1109/ASAP.1991.238909

L. Li, T. Fanni, T. Viitanen, R. Xie, F. Palumbo, L. Raffo, et al., “Low power design methodology for signal processing systems using lightweight dataflow techniques”, 10 2016.

S. Lin, Y. Liu, W. Plishker, et S. S. Bhattacharyya, “A design framework for mapping vectorized synchronous dataflow graphs onto cpu-gpu platforms”, dans *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, série SCOPES ’16. New York, NY, USA : ACM, 2016, pp. 20–29. DOI : 10.1145/2906363.2906374. En ligne : http://doi.acm.org/10.1145/2906363.2906374

F. Maali, P. Ravindra, K. Anyanwu, et S. Decker, “Syrql : A dataflow language for large scale processing of rdf data”, dans *The Semantic Web – ISWC 2014*. Cham : Springer International Publishing, 2014, pp. 147–163.

S. A. Mahmoudi, M. Bagein, et P. Manneback, “Calcul intensif sur GPU : exemples en traitement d ’ images , en bioinformatique et en télécommunication”, *Workshop CIAE’2011, Casablanca*, p. 8p, 2011.

P. Margheritta, “Traçage logiciel d’applications utilisant un processeur graphique”, Mémoire de maîtrise, École Polytechnique de Montréal, 2017, retrieved from https ://publications.polymtl.ca/2838/.

MATLAB version 8.5.0.197613 (R2015a), The Mathworks, Inc., 2015.

R. Mayer, C. Mayer, et L. Laich, “The TensorFlow Partitioning and Scheduling Problem : It’s the Critical Path !” pp. 1–6, 2017. DOI : 10.1145/3154842.3154843. En ligne : http://arxiv.org/abs/1711.01912{%}0Ahttp://dx.doi.org/10.1145/3154842.3154843

A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, et J. Dean, “Device Placement Optimization with Reinforcement Learning”, *Icml*, 2017. En ligne : http://arxiv.org/abs/1706.04972

O. Moindrot, “Triplet Loss and Online Triplet Mining in TensorFlow”, 2018. En ligne : https://omoindrot.github.io/triplet-loss

G. E. Moore, “Cramming more components onto integrated circuits”, *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, Jan 1998. DOI : 10.1109/JPROC.1998.658762

M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, et W. E. Nagel, “Developing scalable applications with vampir, vampirserver and vampirtrace”, dans *PARCO*,

2007.

S. Mysore, B. Mazloom, B. Agrawal, et T. Sherwood, “Understanding and visualizing full systems with data flow tomography”, 2008.

C. Nugteren, “Clblast : A tuned opencl BLAS library”, *CoRR*, vol. abs/1705.05249, 2017. En ligne : http://arxiv.org/abs/1705.05249

NVIDIA CUDA Compute Unified Device Architecture Programming Guide verson 1.0, Nvidia, 2007. En ligne : http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.pdf

CUDA Toolkit Documentation, Nvidia, 2018. En ligne : https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html

D. Orozco, E. Garcia, R. Pavel, J. Arteaga, et G. Gao, “The Design and Implementation of TIDeFlow : A Dataflow-Inspired Execution Model for Parallel Loops and Task Pipelining”, *International Journal of Parallel Programming*, vol. 44, no. 2, pp. 278–307, 2016. DOI : 10.1007/s10766-015-0373-6. En ligne : https://doi.org/10.1007/s10766-015-0373-6

D. K. Osmari, H. T. Vo, C. T. Silva, J. L. D. Comba, et L. Lins, “Visualization and analysis of parallel dataflow execution with smart traces”, dans *2014 27th SIBGRAPI Conference on Graphics, Patterns and Images*, Aug 2014, pp. 165–172. DOI : 10.1109/SIBGRAPI.2014.2

J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, et J. C. Phillips, “Gpu computing”, *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008. DOI : 10.1109/JPROC.2008.917757

H. Perkins, “Cuda-on-cl : A compiler and runtime for running nvidia ; cuda ; c++11 applications on opencl ; 1.2 devices”, dans *Proceedings of the 5th International Workshop on OpenCL*, série IWOCL 2017. New York, NY, USA : ACM, 2017, pp. 6 :1–6 :4. DOI : 10.1145/3078155.3078156. En ligne : http://doi.acm.org/10.1145/3078155.3078156

B. Poirier, R. Roy, et M. Dagenais, “Accurate offline synchronization of distributed traces using kernel-level events”, *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 3, pp. 75–87, Août 2010. DOI : 10.1145/1842733.1842747. En ligne : http://doi.acm.org/10.1145/1842733.1842747

O. Port et Y. Etsion, “Dfiant : A dataflow hardware description language”, 09 2017, pp. 1–4.

P. Rogers, *HSA Overview*, 2015, pp. 7–18. En ligne : http://dx.doi.org/10.1016/B978-0-12-800386-2.00001-8

G. Roquier, M. Wipliez, M. Raulet, J. W. Janneck, I. D. Miller, et D. B. Parlour, "Automatic software synthesis of dataflow program : An mpeg-4 simple profile decoder case study", dans *2008 IEEE Workshop on Signal Processing Systems*, Oct 2008, pp. 281–286. DOI : 10.1109/SIPS.2008.4671776

G. Roquier, E. Bezati, R. Thavot, et M. Mattavelli, "Hardware/software co-design of dataflow programs for reconfigurable hardware and multi-core platforms", dans *Proceedings of the 2011 Conference on Design Architectures for Signal Image Processing (DASIP)*, Nov 2011, pp. 1–7. DOI : 10.1109/DASIP.2011.6136875

S. Rostedt, "Finding Origins of Latencies Using Ftrace", 2009.

D. Schlegel, "Deep Machine Learning on GPUs", pp. 1–6, 2015. En ligne : http://www.ziti.uni-heidelberg.de/ziti/uploads/ce{_}group/seminar/2014-Daniel{_}Schlegel.pdf

S. S. Shende et A. D. Malony, "The Tau Parallel Performance System", *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006. DOI : 10.1177/1094342006064482. En ligne : http://journals.sagepub.com/doi/10.1177/1094342006064482

H. C. D. Silva, F. Pisani, et E. Borin, "A comparative study of SYCL, OpenCL, and OpenMP", *Proceedings - 28th IEEE International Symposium on Computer Architecture and High Performance Computing Workshops, SBAC-PADW 2016*, pp. 61–66, 2017. DOI : 10.1109/SBAC-PADW.2016.19

G. Singer, "The history of the modern graphic processor", 2013. En ligne : https://www.techspot.com/article/650-history-of-the-gpu/

A. Spear, M. Levy, et M. Desnoyers, "Using tracing to solve the multicore system debug problem", *Computer*, vol. 45, no. 12, pp. 60–64, Dec 2012. DOI : 10.1109/MC.2012.191

J. T. Springenberg, A. Dosovitskiy, T. Brox, et M. A. Riedmiller, "Striving for simplicity : The all convolutional net", *CoRR*, vol. abs/1412.6806, 2014. En ligne : http://arxiv.org/abs/1412.6806

J. E. Stone, D. Gohara, et G. Shi, "Opencl : A parallel programming standard for heterogeneous computing systems", *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73,

May 2010. DOI : 10.1109/MCSE.2010.69

G. Stoner, “ROCm : Platform For A New Era of Heterogeneous in HPC and Ultrascale Computing”, 2016. En ligne : https://gpuopen.com/radeon-open-compute-new-era-heterogeneous-in-hpc-ultrascale-computing-the-boltzmann-initiative-delivering-new-opportunities-in-gpu-computing-research/

strace project, strace(1) Linux Manual Pages, 2010, 2010. En ligne : http://man7.org/linux/man-pages/man1/strace.1.html

J. Tompson et K. Schlachter, “An Introduction to the OpenCL Programming Model”, *Digital version available here*, 2012. DOI : 10.1109/MCSE.2010.69. En ligne : http://www.cs.nyu.edu/{~}lerner/spring12/Preso07-OpenCL.pdf

W. W. Wadge et E. A. Ashcroft, *LUCID, the Dataflow Programming Language*. San Diego, CA, USA : Academic Press Professional, Inc., 1985.

F. Wininger, N. Ezzati-Jivan, et M. R. Dagenais, “A declarative framework for stateful analysis of execution traces”, *Software Quality Journal*, vol. 25, 2016.

Mathematica, Version 11.3, Wolfram Research, Inc., 2018.

K. Wongsuphasawat, D. Smilkov, J. Wexler, J. Wilson, D. Mané, D. Fritz, et al., “Visualizing dataflow graphs of deep learning models in tensorflow”, *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 1–12, Jan 2018. DOI : 10.1109/TVCG.2017.2744878